Trabajo Fin de Grado

Arquitecturas de microservicios sobre infraestructuras distribuidas heterogéneas: una prueba de concepto

Microservices architectures on heterogeneous distributed infrastructures: a proof of concept

Autor

Pedro Malo Perisé

Director

José Javier Merseguer Hernaiz

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2020

Agradecimientos

Me gustaría comenzar este proyecto con unas palabras de agradecimiento a mi familia, por haberme dado el apoyo y los recursos necesarios para llegar hasta aquí. También a mis amigos: tanto a los encontrados en la Universidad de Zaragoza y en la Técnica de Luleå, como a los que llevan conmigo desde siempre. Ellos han conseguido que esta etapa haya sido tan valiosa por lo aprendido como por lo vivido.

También a José, el director del proyecto, y al resto de los profesores que han hecho que el camino de estos cuatro años fuese un poco más fácil de recorrer.

Por último, no me gustaría olvidarme de los compañeros de clase que han echado una mano alojando el proyecto y ayudando a que la "arquitectura socializada" fuese una realidad por unas horas.

Resumen

A lo largo de la última década se ha venido observando un aumento sin precedentes de la cuota de mercado del alojamiento *cloud* frente a las soluciones *on premise*. La nube, si bien ha traído consigo un sinfín de ventajas técnicas, ha supuesto también la formación *de facto* de un oligopolio compuesto por las escasas empresas que copan la mayoría de despliegues *cloud*. Esta situación ha creado un equilibrio de poder totalmente asimétrico entre usuarios y proveedores de alojamiento, que sitúa a los segundos en una posición muy ventajosa.

Con el objetivo de intentar revertir dicha situación, en este Trabajo Fin de Grado se propone dar pasos hacia la descentralización del cómputo mediante la que se ha denominado como "arquitectura socializada". En ella, los propios usuarios de un sistema *socializan* las necesidades de cómputo del mismo, cediendo parte de los recursos de sus equipos locales. Si un número suficiente de usuarios participan de la "socialización", el sistema puede llegar a no depender de la nube para ejecutar sus procesos internos.

Para hacer posible este mecanismo, la "arquitectura socializada"se apoya en la virtualización, las comunicaciones asíncronas y los microservicios. Esto da lugar a arquitecturas orientadas a servicios muy ligeros, ejecutables sobre infraestructura heterogénea en un entorno altamente distribuido.

Este proyecto busca probar la viabilidad del modelo arquitectural propuesto mediante una prueba de concepto estudiada de una forma sistemática. Además, propone pautas para el diseño de "sistemas socializados", incidiendo en aspectos como el rendimiento, la seguridad y la distribución.

Abstract

Over the last decade, we have seen an unprecedented increase in the market share of cloud hosting compared to on-premise solutions. Although the cloud has brought with it a myriad of technical advantages, it has also led to a *de facto* oligopoly made up of a few companies that dominate the majority of cloud deployments. This situation has created a totally asymmetric balance of power between users and hosting providers, which settles the latter in a very advantageous position.

In order to try to reverse this situation, this Final Degree Project proposes to decentralize the computation through what has been called a "Socialized Architecture". In it, the users of a system *socialize* its computing needs, giving up part of the resources of their local computers. If big enough number of users participate in a socialization process, the system may no longer depend on the cloud to run its internal processes.

All in all, to make this mechanism possible, the "Socialized Architecture" relies on virtualization, asynchronous communication and microservices. This results in very lightweight service-oriented architectures, executable on heterogeneous infrastructure in a highly distributed environment.

This project wants to test the viability of the architectural model through a proof of concept studied in a systematic fashion. In addition, it proposes guidelines for the design of "socialized systems", focusing on aspects such as performance, security and distribution.

Índice

1. Introducción

Esta es la memoria del Trabajo Fin de Grado "Arquitecturas de microservicios sobre infraestructuras heterogéneas: una prueba de concepto". Este proyecto ha consistido en la implementación de una prueba de concepto para estudiar la viabilidad de la idea que vertebra el proyecto. Queremos utilizar arquitecturas de microservicios para fragmentar sistemas, de nueva creación, en módulos que puedan ser alojados de forma altruista por los usuarios del servicio. Así, la organización detrás del sistema podrá reducir sus necesidades de alojamiento. Tras evaluar la prueba de concepto, se han sintetizado en este documento las conclusiones, una serie de pautas y consideraciones relativas a la implementación de un sistema con la arquitectura propuesta. Esta arquitectura ha sido nombrada como "arquitectura socializada", al distribuirse las necesidades de cómputo entre sus usuarios. Además, se analizan las necesidades a tener en cuenta en un sistema real que hiciese uso de esta arquitectura, se discuten posibles problemas de seguridad y finalmente se valora la viabilidad de la propuesta.

1.1. Justificación

Es un mantra de los últimos años repetir que vivimos en un mundo altamente digitalizado, donde los sistemas informáticos son pieza nuclear de nuestras vidas y que estos juegan un rol totalmente central en nuestro día a día, siendo piezas indispensables para desarrollar vidas dignas, plenas y felices.

No es difícil ver los beneficios que han traído consigo: mejora de innumerables procesos, telecomunicaciones, crecimiento económico y riqueza. Sus defectos tampoco se esconden: destrucción de empleos fruto de la automatización, la brecha digital entre generaciones o los infinitos escándalos sobre privacidad que, cada poco tiempo, inundan los medios de comunicación.

Sin embargo, además de todas estas importantísimas cuestiones, existe otra problemática, otro defecto, mucho menos evidente, y quizá precisamente por esto, más peligroso. Estamos hablando de la dependencia de estos sistemas, totalmente desproporcionada, de alojamiento en la nube. En el año 2018, un 37 % de los despliegues eran *on premise*, esto es, alojados en infraestructura propia de la organización que desarrollaba el sistema. Apenas dos años después, en 2020, un 83 % de los despliegues se realizan sobre soluciones *cloud* [6]: en dos años el uso de la nube ha aumentado su cuota de mercado un 10 %. A este ritmo, en 2025 la nube será la única opción en el mercado. Esta progresión está causada por una buena razón: el alojamiento en la nube ha sido una revolución dentro de la revolución de Internet.

La revolución del *cloud* ha traído un descenso de costes sin precedentes, siendo este el principal motivo por el que las empresas deciden migrar a la nube [7]. Soluciones de escalabilidad, replicación y tolerancia a fallos inalcanzables con aproximaciones *on premise* y una sencillez de uso tal, que hace posible que en pocos *clicks* pueda desplegarse una infraestructura que hasta hace pocos años hubiese supuesto meses de trabajo de varios ingenieros de sistemas. Con todo esto, parece sencillo olvidar la realidad de que alrededor del 74 % del mercado

del alojamiento en *cloud* está en manos de tres empresas: Amazon, Microsoft y Google [36]. Tres empresas copan el mercado de unas soluciones que emplean el 93 % de los negocios. Las matemáticas son claras: de cada 100 empresas que usan la nube, 68 dependerán de datos alojados en un centro de datos de Amazon, Microsoft o Google. Si como hemos dicho esos 68 sistemas son piezas nucleares en la vida de millones de personas, el poder que estamos otorgando a estas empresas es asemejable al de los estados. Sin ir más lejos, el 50 % de los gobiernos usan ya las soluciones *cloud* en sus despliegues [27].

El hecho de que un grupo tan reducido de empresas acapare una proporción tan alta del despliegue de los sistemas informáticos implica un buen número de problemas éticos. Recordemos que el modelo de negocio de los proveedores *cloud* no se basa únicamente en ofrecer capacidad de cómputo, sino también alojamiento a todos los datos de los que se nutren las aplicaciones que soportan.

Si bien existe un amplio marco regulatorio que protege a los usuarios del acceso de terceros a estos datos, no existe ninguna barrera, más allá de la legal, que impida, bien a un operario de estas empresas, bien a la empresa como institución, acceder sin dejar rastro al contenido de una infraestructura sobre la que, en última instancia, tienen control total. La privacidad es quizá la problemática más inmediata, pero no hay que olvidar otras opciones, quizá más difíciles de imaginar, pero igualmente factibles: censura de sistemas o sitios críticos, manipulación de los datos almacenados o alteración del funcionamiento con fines espurios. Esta situación implica un altísimo nivel de centralización del poder. En definitiva, la nube implica la cesión de la soberanía efectiva de los datos y el control del funcionamiento correcto y continuo de nuestros sistemas a grandes corporaciones, sobre las que el ciudadano de a pie carece de mecanismos de control transparentes.

A pesar de este escenario fatalista, quede patente que el hecho de que estas cuestiones son posibles, no quiere decir que necesariamente vayan a suceder. Sin embargo, son una posibilidad, y como tal debe ser considerada, e idealmente, prevenida.

Es evidente que esta prevención no es un motivo de suficiente importancia para renunciar a todas las bondades que ofrece la nube. Los sistemas informáticos son desarrollados, en su mayoría, por empresas que buscan maximizar rentabilidades, y las motivaciones sociales y éticas detrás de la lucha contra oligopolios nunca serán un argumento de peso para renunciar a estas. Es por esto, que si se pretende revertir esta situación, y recuperar el control sobre los sistemas informáticos que requieren de alojamiento, la única opción es hacerlo desde la lucha por la rentabilidad, es decir, **ofrecer alternativas que puedan competir, con el "statu quo"de la nube en términos de costes y conveniencia**.

Ese es precisamente el objetivo de este trabajo fin de grado. Si bien es evidente que ofrecer una alternativa a la nube es una meta descomunal, en este proyecto se pretende hacer una propuesta en tal dirección. Proponemos una arquitectura que permita, con un coste aceptable, despliegues confiables y seguros fuera de la nube, pero sin incurrir en las desventajas de la aproximación *on premise*, tales como altos costes de mantenimiento o la obsolescencia de equipos.

1.2. El concepto

La alternativa que se propone al uso masivo del *cloud* es la "socialización del cómputo". Esto es, repartir las necesidades de cómputo, y eventualmente de almacenamiento, de un sistema informático entre usuarios del mismo que deseen colaborar, empresas que busquen acciones con impacto transformador o cualquier entidad que desee apostar por la reducción de la necesidad de alojamiento en la nube.

El proceso de "socialización" consiste en alojar, de forma altruista o no, una o varias partes del sistema que se desea apoyar en infraestructura propia, dando lugar a entornos de ejecución altamente distribuidos y ejecutado sobre una infraestructura heterogénea.

Es cierto que la idea de colaborar con una causa cediendo capacidad de cómputo local no es en absoluto nueva. Son de sobra conocidos proyectos como el clásico SETI@home [35] (procesado de señales del espacio exterior) o el mucho más reciente DreamLab [12] (cómputo de procesos relacionados con enfermedades como la covid-19, iniciativa de Vodafone en colaboración con el Imperial College de Londres). Ambos proyectos consisten en la instalación de una pieza de software en equipos de propósito general, con la finalidad de que esta ejecute una serie de procesos intensivos en cálculo haciendo uso del *hardware* local. En ambos casos, los usuarios que participan en estos proyectos, lo hacen de forma desinteresada.

Sin embargo, tanto en SETI@home como en DreamLab, las piezas de software local siempre actúan como simples ejecutores esclavos de una tarea concreta muy especializada, orquestada de forma remota y en ambos casos, intensiva en cálculo. En ningún momento actúan como piezas nucleares de un sistema informático mayor, son únicamente músculo hiperespecializado en una tarea.

Lejos de esto, lo que se propone en esta aproximación es la fragmentación de un sistema multipropósito (el *backend* de cualquier sistema moderno) en una serie de servicios independientes, y que cada usuario dispuesto a participar en el proceso de "socialización" aloje de forma local una o varias réplicas de alguno de los servicios que conforman el sistema. Estas réplicas socializadas actuarían como iguales a otras alojadas, bien *on premise*, bien en *cloud* por la organización propietaria del sistema. En función del número de réplicas socializadas en cada momento, la organización podría escalar de forma dinámica el número de réplicas alojadas directamente por ella, e idealmente, con una masa de usuarios suficiente, reducirla a cero logrando así una independencia real de la nube.

Como ya se ha expuesto, la idea de distribuir las necesidades de cómputo de forma altruista está lejos de ser una idea nueva. Sin embargo, el poder hacerlo en los términos comentados, lo posibilita especialmente una serie de tecnologías que han alcanzado su madurez de forma más reciente:

- **Virtualización**. Especialmente, la virtualización ligera basada en contenedores, con tecnologías como Docker [8].
- **Comunicaciones asíncronas**. La superación de patrones como RPC[2],

[2]Remote Procedure Call [31].

dando paso a protocolos de comunicación asíncronos como AMQP [20].

- **Microservicios**. La popularización de diseños basados en arquitecturas orientadas a servicios (SOA), en las que se busca minimizar el alcance de cada uno de los servicios.

A continuación, se discutirá el porqué de la necesidad de que estas tecnologías hayan alcanzado una madurez técnica para poder dar soporte a la aproximación planteada.

1.2.1. Virtualización

Una diferencia notable entre las aproximaciones de cómputo distribuido, y la aquí propuesta radica en el modo de desarrollo. Para que la "socialización de servicios"sea una alternativa competitiva al desarrollo y despliegue clásico, el desarrollo de estos no puede suponer una gran diferencia con respecto al desarrollo tradicional. Resultaría inadmisible exigir que el desarrollo de un servicio socializable tuviese que ser realizado con un lenguaje de programación específico, un *framework* concreto o respetando cualquier requisito no funcional derivado del entorno en el que fuese a ser alojado, tal como es el caso de proyectos como SETI@home, donde las tareas a ejecutar son totalmente dependientes del entorno provisto por la aplicación cliente.

Esta premisa, si no se opta por una solución basada en la virtualización, parece muy difícil de respetar, ya que convertiría la instalación de la réplica local en un proceso complejo y propenso a los errores. Todas las dependencias y configuraciones necesarias deberían ser instaladas de forma manual por el usuario que desee colaborar con el proyecto, con el nivel de pericia técnica que esto requiere. Además, ejecutar el servicio de forma nativa siempre podría dar lugar a incompatibilidades de versiones o sistemas. En definitiva, renunciar a la virtualización implicaría renunciar también a un número elevado de réplicas socializadas.

La solución a estos problemas pasa por la virtualización, especialmente, la virtualización ligera basada en contenedores. La virtualización basada en contenedores (o a nivel de sistema operativo) provee de entornos estancos de ejecución que usan directamente el sistema operativo del equipo anfitrión. Esto permite un alto grado de aislamiento, a la par que apenas supone una sobrecarga al sistema, ya que no es necesario emular el sistema virtualizado al completo. Para el propósito de la "socialización de servicios", se trata de la solución ideal: 1) provee de una forma sencilla de empaquetar software junto a sus dependencias y configuraciones; 2) ofrece entornos aislados de ejecución; y 3) apenas implica una sobrecarga en el sistema mayor que la que derivaría ejecutar el servicio de forma nativa. Por último, el software de gestión de contenedores, como por ejemplo Docker, está ampliamente extendido, es fácil de usar y es de código abierto, lo que posibilita un uso más amplio.

1.2.2. Comunicación asíncrona

Una de las contrapartidas más evidentes de la "socialización de servicios" es lo inestable de la infraestructura. En los ambientes *cloud*, el control sobre esta es total: su presencia y normal funcionamiento están garantizados con SLA[3] de más del 99 % gracias a altos grados de replicación. El hardware sobre el que se ejecuta es homogéneo. Adicionalmente, las empresas proveedoras de servicios en la nube invierten enormes sumas de dinero en infraestructuras como cables submarinos o enlaces de fibra óptica que minimizan los tiempos de latencia entre sus centros de datos. Todo esto redunda en que los despliegues en la nube gocen de una estabilidad y seguridad difícilmente alcanzable en otros contextos. Este es precisamente uno de los motivos del éxito del modelo *cloud*.

Al depender en el "modelo socializado"de una infraestructura tan heterogénea, no sólo no se disfruta de las ventajas de la nube, sino que además una nueva gama de posibles problemas aparece. Algunos de ellos son los siguientes:

- **Tiempos de latencia no homogéneos**. Un mismo servicio puede estar replicado tanto en una *Raspberry Pi* conectada a una red doméstica como en el centro de datos de una universidad.
- **Inestabilidad de la red**. Las redes domésticas tienen políticas de calidad de servicio (QoS) muy distintas de las redes *ad hoc* de los centros de datos. Además, son vulnerables a verse afectadas por el uso que los demás usuarios hagan del ancho de banda. Por ejemplo, un usuario que aloja una réplica de un servicio socializado, puede decidir consumir un servicio de vídeo en *streaming* al mismo tiempo.
- **NATs e IPs privadas**. Apenas ninguna red doméstica actual posee una dirección IP pública estática. La mayoría de ellas se encuentran tras una NAT[4] y sus direcciones asociadas compartidas pueden cambiar a criterio del proveedor de servicios de Internet (ISP). Esto hace que cualquier comunicación entrante a una instancia de un "servicio socializado"pueda no ser posible al desconocer los demás servicios cómo alcanzar tal réplica.

Esta serie de problemas lleva a descartar *a priori* protocolos de comunicación entre servicios síncronos, como puedan ser REST[5][26] o RPC. Este tipo de protocolos dependen para un funcionamiento óptimo de tiempos de latencia acotados, cierto grado de estabilidad en la red y conocimiento de la dirección de envío de los mensajes, y nada de esto puede ser garantizado en un entorno tan heterogéneo como este.

Estas restricciones llevan a considerar los protocolos de comunicación asíncronos. Estos patrones de comunicación se caracterizan por la no coincidencia temporal entre dos actores a la hora de establecer comunicación. El primero de ellos envía un mensaje siguiendo una aproximación *fire-and-forget* y el segundo lo recibirá, y responderá si procede, en un momento indeterminado posterior al

[3] Service Level Agreements.
[4] Network Address Translation.
[5] Transferencia de Estado Representacional.

envío del primer mensaje. De esta forma, el primer actor no queda bloqueado a la espera de la respuesta del segundo, y puede continuar su flujo de ejecución normal.

Mientras que en el mundo físico este tipo de comunicación se muestra en ejemplos como el correo postal o la televisión, en el campo de la informática su presencia más clara se encuentra en los sistemas de paso de mensajes [31]. Si combinamos este patrón de comunicación con el concepto de *broker* de mensajes[6] la problemática expuesta queda resuelta:

- Los tiempos de latencia entre servicios pasan a ser irrelevantes, ya que en el momento que se adopta el patrón asíncrono, los emisores de los mensajes continúan con su flujo de trabajo normal hasta que, si es necesario, llegue una respuesta. Esto se traduce en un flujo de control reactivo: en el momento que llegue una eventual respuesta, interrumpirán su ejecución habitual, atenderán la respuesta, y volverán de nuevo a su flujo normal.

- La inestabilidad de la red pasa a ser un problema menor, siempre y cuando el *broker* de mensajes se mantenga accesible: si un emisor envía un mensaje, y en ese momento una de las posibles réplicas receptoras se desconecta de la red, otra tomará el mensaje de la cola de mensajería en su lugar, pudiendo completarse la comunicación. Si por ejemplo una partición de red dejase todas las réplicas sin comunicación con el *broker*, los mecanismos de fiabilidad de este persistirían el mensaje de forma local, hasta que alguna réplica estuviese lista para procesarlo.

- El descubrimiento de las direcciones de los servicios deja también de ser un problema, ya que al actuar el *broker* como agente de entrega centralizado, son los servicios los que inician la comunicación con él. El *broker*, al estar alojado en una dirección estática conocida, es siempre fácilmente localizable por los servicios, con independencia de si se encuentran tras una NAT. Además, el proceso de cambio de IP pública pasa a ser totalmente transparente.

El uso de un *broker* de mensajería proporciona otras ventajas sobre la comunicación punto a punto tradicional:

- Permite establecer políticas de entrega, *timeouts* o seguridad centralizadas, teniendo estas que ser configuradas en un único lugar.

- Reduce la superficie de exposición a posibles ataques, en lugar de tener que securizar los *endpoints* de cada servicio de forma individual.

- Ofrece un lugar centralizado para la recogida de *logs* y métricas de uso.

- Existen multitud de implementaciones y protocolos libres para ejecutar el *broker*. Entre ellos destaca, como se comentará en la Sección 4, RabbitMQ [37].

[6] Actor central que encamina los mensajes y provee de mecanismos de fiabilidad y garantía.

A pesar de que las ventajas de emplear un *broker* asíncrono de mensajería para posibilitar la socialización de servicios son evidentes, existen también algunos inconvenientes que deben ser tenidos en cuenta:

- El *broker* pasa a ser un punto único de fallo del sistema: si el *broker* dejase de funcionar, el sistema entero lo haría también. Sin embargo, esta es una problemática bien estudiada, y la mayoría de implementaciones ofrecen mecanismos de replicación y "clusterización"que minimizan este riesgo.
- Ejecutar el *broker* supone un coste de infraestructura adicional, que en el caso de las comunicaciones síncronas punto a punto no existe. Este coste es aún mayor si se aplica el grado de replicación necesario para mitigar el riesgo anterior.

1.2.3. Arquitecturas de microservicios

La "socialización de servicios"pasa en primer lugar por dividir el sistema en partes menores, que más adelante puedan ser "contenerizadas" y alojadas por terceros.

Para que este proceso sea viable, las partes resultantes deben ser del menor tamaño posible, ya que de otra forma, el proceso de socialización pierde uno de sus incentivos principales para los usuarios colaboradores: el impacto mínimo de alojar una réplica del servicio en el *hardware* cedido altruistamente. Resulta evidente que nadie va a estar dispuesto a alojar un contenedor en su equipo, si el hecho de alojarlo consume la mayoría de los recursos de este. Es por esto, que minimizar el tamaño de las réplicas, tanto en espacio como en demanda de recursos, es esencial.

Si tomamos los requisitos de particionado de software y minimización del tamaño de estas particiones, la solución arquitectural es clara: microservicios.

En *Building microservices*[19], Sam Newman los describe como "una aproximación a de los sistemas distribuidos que promueve el uso de servicios de granularidad fina con sus propios ciclos de vida, los cuales colaboran de forma conjunta". Las arquitecturas de microservicios (MSA) se caracterizan por un desarrollo fragmentado, en el que los distintos servicios que las componen se comunican entre sí, buscando maximizar la cohesión y minimizar el acoplamiento. Esto, entre otros, se consigue reduciendo el alcance de los servicios, restringiéndolos a dominios muy concretos, especializados en una serie de casos de uso concreto. Dicha reducción del alcance redunda en servicios pequeños, que es, como se ha discutido, lo deseable para el proceso de socialización.

Es cierto que el objetivo principal de las arquitecturas de microservicios no es la reducción por la reducción del tamaño del software resultante. Dicha reducción es, en todo caso, una consecuencia de las prácticas adoptadas y de la propia fragmentación del sistema: las partes individuales de un monolito siempre serán menores al monolito en su conjunto. A pesar de ello, se ha considerado dicha reducción un factor determinante para optar por este patrón de diseño arquitectural. El requisito de minimizar el impacto de la socialización de réplicas en la infraestructura cedida hace que sea la única opción aceptable.

Al margen de esta cuestión, el uso de microservicios trae consigo una serie de ventajas, que satisfacen otras necesidades del proceso de socialización.

- **Resiliencia**. Los sistemas basados en microservicios son más tolerantes a fallos, ya que la eventual caída de un servicio no causa un fallo global en el resto del sistema. Los procesos que no dependen de dicho servicio pueden seguir siendo ejecutados con total normalidad. Esto resulta muy interesante en el contexto de la socialización, ya que la infraestructura es mucho más propensa al fallo que en entornos *cloud* u *on premise*.

- **Escalabilidad**. La arquitectura de microservicios es mucho más escalable que la monolítica. En caso de detectarse una sobrecarga en ciertos casos de uso, basta con escalar horizontalmente los servicios relacionados con dichos casos de uso, pudiendo el resto mantener su cantidad de réplicas. De nuevo, esta propiedad es muy deseable en nuestro contexto, ya que ante un posible número bajo de réplicas de cierto servicio, resulta más fácil reaccionar por parte de la organización.

- **Simplificación en el despliegue**. El despliegue individual de las instancias de un servicio siempre será mucho más sencillo que el de un gran monolito: las dependencias serán menores y más ligeras, y las configuraciones necesarias más simples. Esto, como se comenta en la sección de virtualización, resulta fundamental para incentivar la instalación local de réplicas.

Además de estos beneficios concretos, los microservicios tienen otras ventajas de las que se beneficia cualquier sistema. Por ejemplo, mejoras organizativas en la gestión del proyecto, la posibilidad de emplear las tecnologías que se adapten mejor a cada caso de uso sin que ello suponga comprometerse a usarlas en el resto del sistema o facilitar el proceso de *onboarding* de nuevos desarrolladores.

A pesar de todas las ventajas, esta arquitecturas implica también una serie de inconvenientes que deben ser tenidos en cuenta.

- Las arquitecturas de microservicios son mucho más complejas que las monolíticas. Los procesos de comunicación y el nuevo abanico de casuísticas de fallo que traen consigo, hacen que, en general, desarrollar microservicios sea más difícil que desarrollar monolitos.

- Las comunicaciones entre distintos microservicios, bien mediante REST, bien mediante paso de mensajes, siempre tendrán mayores latencia que las llamadas al sistema de un monolito.

- Someter a *testing* un sistema de microservicios reviste una mayor complejidad que someter un monolito.

Por último, cabe destacar que las arquitecturas de microservicios están lejos de ser una tecnología reciente. El nombre apareció por primera vez en el año 2005, en una conferencia ofrecida por Peter Rodgers [28], pero el concepto detrás

de él era conocido hace tiempo. A pesar de ello, lo que hace que sea una pieza clave para la "socialización de servicios" es la enorme popularidad que han adquirido en los últimos años. Una buena prueba de ello es la evolución que desde 2012 el concepto de microservicios presenta en el radar de ThoughtWorks [33]. El hecho de que esta tecnología se haya popularizado tanto posibilita encontrar desarrolladores competentes en esta técnica y que los conceptos subyacentes sean familiares en los foros técnicos. Esto es fundamental para que la socialización de servicios pueda llegar a ser una alternativa real a la nube.

1.3. Metodología

La metodología planteada para el proyecto se basa en una prueba de concepto, esto es, la elaboración de un prototipo funcional que permita valorar con mayor criterio la viabilidad de la "arquitectura socializada". Se ha optado por esta aproximación, ya que un abordaje meramente teórico sería poco adecuado por la escasez de fuentes al respecto sobre un concepto tan disruptivo.

Además, el hecho de enfrentarse con una implementación real hace que se tomen en consideración muchos aspectos derivados del propio proceso de implementación, que en otras circunstancias, pasarían desapercibidos.

Los detalles del caso de uso concreto serán desarrollados más adelante, en el apartado correspondiente, pero sus principales características pasan por ser de gran simplicidad conceptual, a la par que cuenta con un buen número de casuísticas a considerar.

En cuanto a la metodología seguida en el proceso de implementación, se ha seguido una aproximación iterativa. Una vez establecidos los casos de uso en los que se sustenta la prueba de concepto, se ha procedido a desarrollar una serie una serie de sub-pruebas de concepto, con el doble objetivo de familiarizarse con las tecnologías usadas y probar los conceptos más básicos como el patrón de comunicación. Después de esto, se procedió a implementar un primer prototipo, en el que se primó un desarrollo rápido. A continuación, se refactorizó el prototipo inicial, extrayendo utilidades comunes en una librería externa reutilizable y aplicando conceptos de Domain Driven Design (DDD) [9]. Finalmente, siguiendo criterios de buenas prácticas y calidad de software se rediseñó parte del proyecto. Esta última refactorización es la que ha dado lugar a la versión final de la prueba de concepto.

Tras la implementación, se ha procedido a evaluar las prestaciones de la prueba de concepto haciendo uso de unas métricas sencillas que serán expuestas más adelante. Sin embargo, dada la sencillez del caso de uso, estas métricas han resultado poco representativas, más allá de probar empíricamente la viabilidad de la implementación.

Finalmente cabe destacar que la mayor parte del valor que aporta este trabajo procede de recoger una serie de pautas, consideraciones y propuestas sobre la implementación de sistemas basados en "arquitecturas socializadas". Estas constituyen el grueso de este documento, y se han ido acumulando fruto del proceso de implementación, donde algunas de ellas se han revelado como aciertos y otras errores.

2. El caso de prueba

2.1. Visión general

Resulta evidente que la implementación de un sistema, con las características del expuesto en la sección anterior, y a una escala realista es enorme, más aún para un proyecto realizado por una única persona. El reto es mayor si se plantea siguiendo la "arquitectura socializada", ya que esto implicaría al menos: el desarrollo de nuevas tecnologías, considerar un buen número de supuestos, tener en cuenta múltiples detalles de implementación ligados al sistema concreto, desarrollar microservicios en diferentes dominios, así como muchos otros aspectos.

Es por esto que la opción que se ha encontrado más razonable sea implementar un sistema a pequeña escala, pero que se vea afectado por el mayor número posible de casuísticas: comunicación entre servicios, entidades compartidas, requisitos de escalabilidad o el uso de eventos de dominio.

En base a esto se ha diseñado **Circle**, el sistema que será implementado como caso de prueba. Circle es una red social que soporta una serie de funcionalidades muy básicas, pero que en conjunto conforman un sistema funcional. En esencia, es asemejable a una versión muy básica de Twitter [34]: usuarios registrados (que poseen un perfil) pueden realizar publicaciones de texto. Estas publicaciones pueden ser apreciadas con un *like* por otros usuarios. Además, es posible obtener listas con las publicaciones más valoradas o más recientes, así como de los usuarios más populares. Por último, Circle incorpora un sistema de notificaciones y suscripciones, por el cual se envían *emails* a los usuarios cuando se dan ciertos eventos. Es posible por tanto, suscribirse a un usuario y recibir una notificación cada vez que este realiza una publicación. Los requisitos formales del sistema serán incluidos más adelante.

El producto final es accesible mediante una API GraphQL[10] que da soporte a los distintos casos de uso expuestos. Evidentemente incluye también los servicios que los ejecutan, los mecanismos de persistencia y la infraestructura necesaria para que el sistema sea operativo. No se ha implementado interfaz de usuario ya que se ha considerado fuera del alcance del proyecto.

El dominio escogido, en apariencia sencillo, es lo suficientemente rico para dar pie a un amplio abanico de casos de uso. Además, resulta muy adecuado al ser fácilmente fragmentable en subdominios que puedan ser gestionados por microservicios independientes sin un exceso de comunicación cruzada, aunque esta se da en casos suficientes como para haber tenido que ser considerada.

Algunos aspectos que la riqueza del sistema obliga a considerar son los siguientes.

- Necesidad de garantizar la consistencia entre entidades gestionadas por distintos microservicios. Por ejemplo, para dar soporte al caso de uso de obtener a los usuarios con más publicaciones, es necesario mantener de forma consistente una relación entre el microservicio de usuarios y el de contenidos. Este requisito fuerza a implementar mecanismos complejos (reacción a eventos de dominio) para asegurar la integridad.

- Interacción con servicios externos en base a eventos internos. Es por ejemplo, el caso del envío de *emails* cuando se publican contenidos a los que se está suscrito.
- Consultas que requieran de la agregación de información que reside en servicios diferentes. Por ejemplo, al obtener el perfil de un usuario, el cual incluye una lista de sus publicaciones.
- Modelo de escalamiento sencillo. Debe ser fácil añadir réplicas de un mismo servicio sin excesivas complicaciones.

2.2. Historias de usuario

A continuación se adjunta una relación de las historias de usuario definidas formalmente que cubren toda la funcionalidad de **Circle**.

2.2.1. Usuarios

- **HU1 -** Como usuario, quiero poder registrarme en el sistema aportando un nombre de usuario y una dirección de *email*
- **HU2 -** Como usuario, quiero poder modificar mi perfil de usuario modificando el *email*
- **HU3 -** Como usuario, quiero poder consultar el perfil de otros usuarios (identificador, *email* y publicaciones) a partir de su nombre de usuario.
- **HU4 -** Como usuario, quiero poder consultar una lista de todos los usuarios registrados en el sistema.
- **HU5 -** Como usuario, quiero poder consultar los perfiles de los N usuarios con más publicaciones.
- **HU6 -** Como usuario, quiero poder consultar los perfiles de los N usuarios con más suscriptores.

2.2.2. Contenidos

- **HU7 -** Como usuario, quiero poder añadir publicaciones formadas por un título y un cuerpo.
- **HU8 -** Como usuario, quiero poder consultar una lista con las N publicaciones más recientes.
- **HU9 -** Como usuario, quiero poder consultar una lista con las N publicaciones con más *likes*.
- **HU10 -** Como usuario, quiero poder dar *likes* a las publicaciones de otros usuarios.

2.2.3. Suscripciones

- **HU11 -** Como usuario, quiero poder suscribirme a otros usuarios.

2.2.4. Notificaciones

- **HU12 -** Como usuario, quiero poder recibir una notificación por *email* cuando un usuario al que estoy suscrito añada una nueva publicación.
- **HU13 -** Como usuario, quiero poder recibir una notificación por *email* cuando otro usuario se suscriba a mis publicaciones.
- **HU14 -** Como usuario, quiero poder recibir una notificación por *email* cuando otro usuario dé *like* a alguna de mis publicaciones.

3. Arquitectura

A continuación se procede a analizar la arquitectura planteada para la solución del caso de prueba, haciendo hincapié en los detalles más relevantes como el papel de la asincronía, el patrón de diseño CQRS[7] y la estructura de colas.

3.1. Visión general

La esencia de la "arquitectura socializada"pasa por distribuir parte del cómputo de un sistema entre máquinas domésticas. A la hora de diseñar la arquitectura de un sistema que implemente este patrón, se deben tener en cuenta algunas consideraciones que ya se han comentado. En esta prueba de concepto se han omitido algunos aspectos por tener un tratamiento especialmente complejo, pero la arquitectura que la resuelve ha sido desarrollada con esos principios en mente.

En primer lugar, se ha distribuido toda la lógica de negocio del sistema en varios servicios independientes. Estos han sido el servicio de usuarios (*users-service*), el de contenidos (*contents-service*) y el de suscripciones (*subscriptions-service*). Estos servicios actúan únicamente como ejecutores de la lógica de **Circle**. Carecen de estado interno y son replicables desde el diseño, es decir, es posible ejecutar varias réplicas de un mismo servicio sin configuración adicional. De ellos se esperan cinco funciones muy bien definidas:

- Atender las órdenes demandas a la API, aceptarlas o rechazarlas una vez validados los parámetros, y en su caso ejecutarlas.
- Resolver las consultas demandadas por la API, aceptarlas o rechazarlas una vez validados los parámetros, y en su caso responder con la resolución de la consulta.
- Emitir eventos de dominio como resultado de los comandos que los disparen.
- Escuchar eventos de dominio emitidos por otros servicios y reaccionar a ellos.
- Interactuar con el servicio de persistencia cuando algún caso de uso lo requiera.

En cuanto a los distintos servicios, cabe destacar que carecen de mecanismos de persistencia locales. En su lugar, todas las réplicas de un mismo servicio interactúan con una única base de datos alojada remotamente. Esta aproximación permite que cada servicio tenga su propio modelo relacional independiente de los demás. Sin embargo, que todos las instancias compartan acceso a una base de datos externa no es lo más deseable en el marco de la "arquitectura socializada", ya que obliga a seguir dependiendo de la nube para alojar la persistencia con fiabilidad. A pesar de ello, en esta prueba de concepto se ha optado por

[7] Command Query Responsibility Segregation [16, 17].

ello, ya que, como se comentará en la sección de trabajo a futuro, socializar también la persistencia de datos tiene un buen número de implicaciones de gran complejidad que escapan al ámbito de este proyecto. En la figura 1 se incluye el diagrama de componentes del servicio de usuarios, análogo a cualquier servicio de la prueba de concepto.

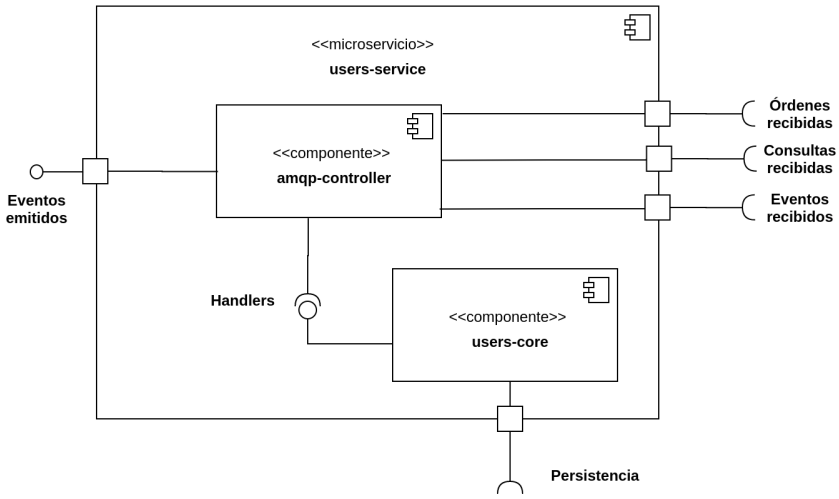

Figura 1: Diagrama de componentes del servicio de usuarios

Como puede observarse, cada servicio está formado por dos componentes principales. Por un lado encontramos el **amqp-controller**, cuya responsabilidad es la de comunicarse con los demás servicios del sistema. Para ello, este componente reacciona a órdenes, consultas y eventos a los que esté suscrito, además de publicar eventos de dominio de acuerdo a la lógica de negocio de los casos de uso ejecutados. En el anexo A puede consultarse la documentación de cada puerto de este componente.

Por otro lado encontramos el núcleo o **core** del servicio, el componente que encapsula toda la lógica de negocio de la que este servicio es responsable. Estos casos de uso, mediante un registro previo de *callbacks* en el controlador AMQP, se ejecutan cuando resulta preciso. Como puede apreciarse en el diagrama, este componente es el responsable de interactuar con la persistencia del sistema. Los detalles de cómo sucede esto serán desarrollados en el capítulo de implementación.

Otro detalle importante a considerar es la forma de interaccionar entre los clientes del sistema y los servicios. Una primera solución podría consistir en que los clientes se dirigiesen directamente a los servicios, pero esta aproximación presenta varios problemas.

- Los clientes están obligados a conocer la dirección de los servicios para poder dirigirse a ellos. Dado que se aspira a que un buen número de los

servicios estén alojados en máquinas conectadas a redes domésticas, esto se tornaría muy complicado. La única solución sería depender de un servicio de DNS (Domain Name System) dinámico que detectase los cambios de IP pública.

- Los mecanismos de balanceo de carga dejan de ser transparentes al usuario. Si se desea que la carga rote entre múltiples instancias de un mismo servicio, la única solución es la implementación de balanceo de carga mediante DNS Round Robin [13]. Esta es una solución bastante limitada al basarse únicamente en un algoritmo "todos contra todos", en el que no se contempla, por ejemplo, el coste de atender cada petición.

- La agregación de información procedente de distintos servicios deja de ser transparente al cliente, ya que pasa a ser el responsable de realizarla.

- Una vez decidido el uso del protocolo AMQP, la comunicación entre servicios y cliente debería ser necesariamente mediante este protocolo. Esto tendría un impacto directo en la interoperabilidad de la API, ya que AMQP no es en absoluto la opción más habitual para comunicar clientes de APIs con el *backend*.

Es debido a estos motivos que se decidió incluir un elemento nuevo en la arquitectura: una puerta de enlace o API *Gateway*. Se trata de un patrón de diseño bastante habitual en arquitecturas de microservicios. Un API *Gateway* es un servicio, ajeno al dominio, que tiene como cometido dar solución a los problemas anteriores: gestiona el balanceo de carga y la resolución de nombres, agrega la información de ser necesario y traduce el protocolo de comunicación. Además, en términos de seguridad, tiene la ventaja adicional de reducir la superficie de exposición del sistema y de poder actuar como entidad de autenticación.

El uso de un API *Gateway* es muy interesante para el caso de prueba, pero implica al mismo tiempo una serie de desventajas que deben ser consideradas: se convierte en otro punto único de fallo del sistema y requiere de infraestructura adicional para ser alojado. Al igual que con el soporte de persistencia, en este caso de prueba se propone que el API *Gateway* sea alojado directamente por la organización y no sea socializado. Esto es así debido a las implicaciones de seguridad y rendimiento que traería alojar un servicio tan crítico en máquinas no optimizadas.

Como resumen de la visión general de la arquitectura, en la figura 2 se adjunta el diagrama de despliegue del sistema. Se ha escogido esta vista al considerarse mucho más representativa que otras más habituales como el diagrama de clases o de componentes.

3.2. Elementos destacables

A continuación, se profundiza en algunos aspectos de la arquitectura que se han considerado de especial relevancia.

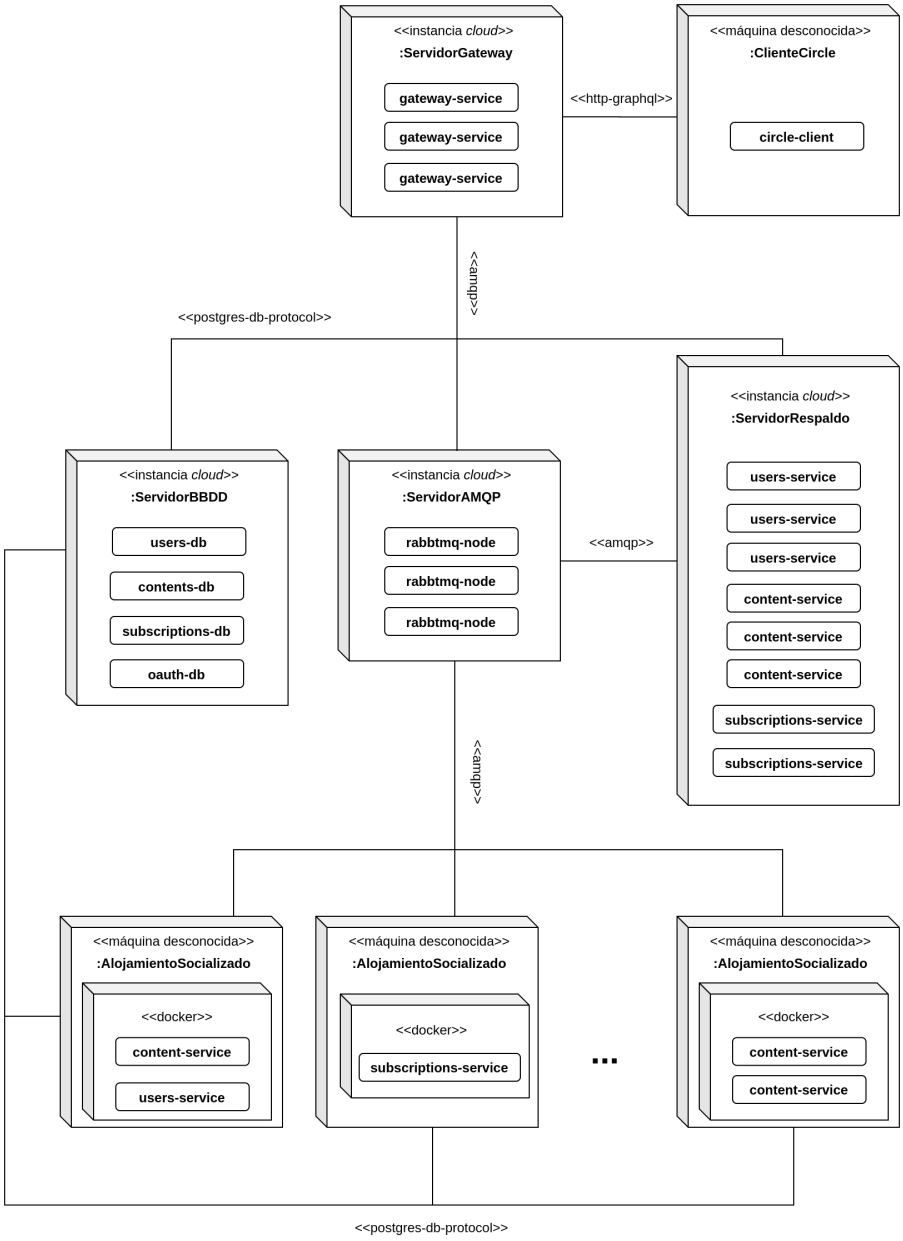

Figura 2: Vista de despliegue de la arquitectura propuesta

3.2.1. Modelo asíncrono

Una de las características más relevantes de la arquitectura es el papel central de la asincronía, es decir, el uso de patrones de comunicación en los que no

es necesario que los interlocutores coexistan temporalmente. Como ya se ha comentado, la justificación de esta aproximación radica en la escasa fiabilidad que proporciona la infraestructura heterogénea sobre la que se van a ejecutar los sistemas socializados.

El hecho de usar mecanismos de comunicación asíncronos convierten el sistema en un sistema reactivo. Mientras que en un sistema tradicional la comunicación entre servicios sucede punto a punto (si un servicio desea hablar con otro se dirige directamente a él), en uno reactivo cada servicio actúa como un componente independiente, que se limita a "reaccionar" a los eventos que suceden en el entorno del sistema, en función de la lógica de negocio.

Un buen ejemplo de esta aproximación sería cómo lleva a cabo el sistema la historia de usuario **HU7**. Mientras que en un sistema tradicional el servicio de contenidos se comunicaría directamente, por ejemplo, mediante una llamada REST, con el servicio de usuarios para incrementar el número de publicaciones del autor, en un sistema reactivo esto no sucedería así. Ante el envío de la publicación, el servicio de contenidos se limitaría a publicar un evento notificando la creación de un nuevo *post*. Cualquier otro servicio interesado en dicho evento estaría suscrito a él, y reaccionaría de la forma especificada por la lógica de negocio. De esta forma, se maximiza el desacoplamiento entre servicios. Además, la comunicación directa entre servicios deja de ser necesaria, por lo que los problemas derivados de latencias no homogéneas pierden parte de su importancia.

De esta aproximación derivan las denominadas "Arquitecturas dirigidas por eventos", conocidas bajo las siglas EDA[8] en inglés.

3.2.2. Separación de responsabilidades

Otra de las características más destacables del sistema es haber seguido el patrón de diseño Command Query Responsibility Segregation [17] (CQRS). Esta aproximación se caracteriza por una clara distinción entre las dos posibles formas de interacción que el sistema ofrece: *commands* (órdenes) y *queries* (consultas). Los *commands* se caracterizan por ser acciones que fuerzan un cambio de estado en el sistema y no devuelven nada. Por otro lado, las *queries* son operaciones que no alteran el estado del sistema y siempre devuelven información. Órdenes y consultas toman caminos bien diferenciados en el sistema, con modelos de procesamiento totalmente segregados.

Además de *commands* y *queries*, el otro concepto fundamental de CQRS es la noción de *bus*. Para la distribución de órdenes y consultas CQRS propone el uso de sendos *buses* a los cuales se suscribirían una serie de *handlers*. Esta idea suele implementarse partiendo de una aproximación monolítica, en la cual el *bus* se limita a gestionar la llamada de funciones locales que representan los *handlers*. En nuestro contexto, basado en microservicios, esta solución no es posible. En su lugar, se propone una aproximación en la cual el API *Gateway* publica órdenes y consultas empaquetadas dentro de mensajes AMQP, a los

[8] Event Driven Architectures [32].

cuales los servicios pueden suscribirse. La función ocupada de reaccionar a la recepción de un mensaje cumpliría un rol análogo al de los *handlers* clásicos. Esta aproximación puede resumirse a alto nivel en el diagrama de la Figura 3

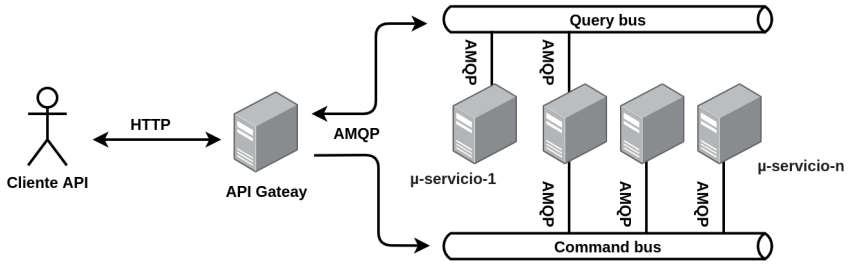

Figura 3: Vista de alto nivel del patrón CQRS aplicado a microservicios

Como explica Martin Fowler en el artículo de referencia [17], el uso de CQRS trae consigo una serie de ventajas. Entre ellas se encuentra la implementación de sistemas más ricos, lejos de los modelos anémicos a los que suele llevar el diseño de sistemas CRUD[9] puros. Además, como también explica Fowler, los sistemas que aplican CQRS suelen tener un mejor rendimiento que los que no.

El uso de CQRS implica también un aumento de la complejidad en los sistemas, ya que fuerza al desarrollador a pensar de una forma diferente a la habitual. Como ejemplo de ello, resulta muy ilustrativo un hipotético caso de uso de inicio de sesión. En el momento que iniciar sesión en el sistema cambia el estado de este, esta acción debe ser necesariamente un *command*. Por tanto, no debería devolver nada, más allá de si la orden ha sido aceptado para su procesamiento en base a mecanismos de validación de los parámetros. La forma de informar al usuario sobre el resultado del inicio de sesión pasa por mecanismos de suscripción al resultado del comando, que de forma asíncrona, le notifiquen el resultado con la información adecuada. Esto se desarrollará, más adelante, en el capítulo de implementación.

Como puede extraerse del ejemplo anterior, el coste de adoptar CQRS es alto. Sin embargo, ha decidido optarse por él en base al argumento del rendimiento y los principios intrínsecamente asíncronos que lo rigen. Es cierto que el caso de prueba propuesto está lejos de ser intensivo en cálculo o ser crítico al nivel de necesitar respuestas inmediatas, pero debe tenerse en cuenta lo incierto de la infraestructura que lo va a ejecutar. Por tanto, se ha decidido asumir el peor escenario posible y dar por hecho que el rendimiento debe optimizarse al máximo. En cualquier caso, dada la sencillez del caso de prueba, esta decisión responde más a establecer una pauta de diseño que a un requisito real del prototipo.

[9] Create, Read, Upadate, Delete.

3.2.3. Estructura de colas

Un aspecto de la arquitectura que merece ser reseñado, dado el uso intensivo que se hace del modelo asíncrono, es la estructura de colas planteada dentro del sistema. Cabe recordar que el elemento que hace posible la comunicación asíncrona entre los diferentes elementos del sistema son las colas de mensajes. Las colas de mensajes son estructuras de datos, gestionadas por el *broker* en las cuales es posible publicar y consumir mensajes con una política FIFO (*First In First Out*). Estos mensajes son objetos serializados que representan órdenes, consultas y eventos. A la hora de diseñar el sistema de colas, se tomaron las siguientes decisiones:

- Cada emisor de órdenes, consultas y eventos publica estos objetos en una centralita. A dicha centralita se suscribirán, usando cada uno una cola diferente, los servicios interesados en dichas órdenes, consultas o eventos. De esta forma se consigue que distintos consumidores no "se roben"los objetos de las colas entre sí, ya que estos se replican tantas veces como colas, y por tanto servicios, haya suscritos a dichos objetos.

- Todas las instancias de un mismo servicio utilizarán la misma cola para un tipo concreto de orden, consulta o evento. De esta forma las distintas réplicas sí compiten por el objeto, y sólo una lo consigue en cada ocasión. De esta forma escalar un servicio es tan simple como añadir más réplicas suscritas a la misma cola, sin necesidad de ninguna configuración de balanceo de carga. Por este motivo se puede afirmar que la arquitectura es escalable desde el diseño.

- Las instancias de los servicios que esperen una respuesta como resultado de una publicación (en el caso de prueba esto sucede únicamente en el API *Gateway* con los objetos de tipo consulta) tendrán cada uno una cola propia para respuestas. Junto con la publicación de la consulta, las réplicas adjuntarán el nombre de la cola a la que debe responder el servicio ejecutor de la misma. Así, únicamente la réplica del API *Gateway* que ha solicitado la consulta obtendrá una respuesta, y podrá hacérsela llegar al cliente simulando sincronismo.

Esa estructura de colas queda representada a alto nivel en el diagrama de la Figura 4.

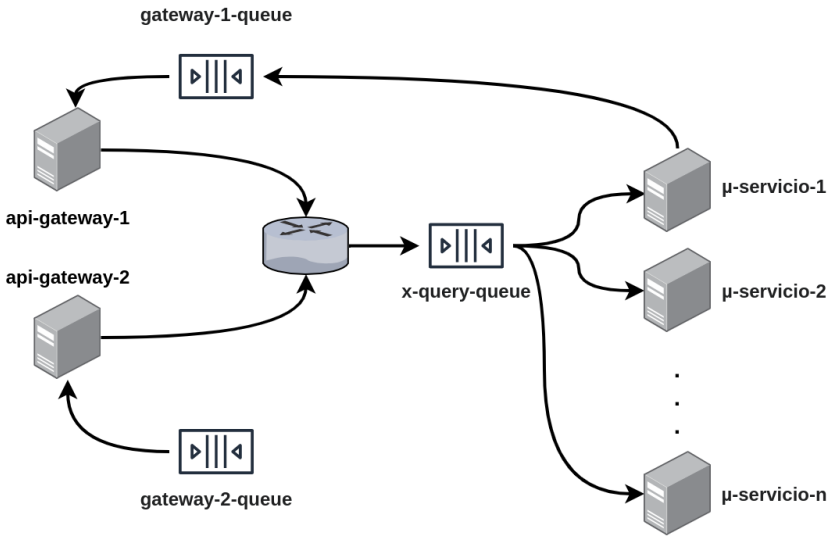

Figura 4: Vista de despliegue de la arquitectura propuesta

4. Cuestiones de implementación

En este capítulo se analizarán aquellos aspectos que escapan a las cuestiones de diseño arquitectural. Son por tanto decisiones que a veces abordan cuestiones meramente de implementación pero en otras implican decisiones de diseño, aunque no arquitecturales. Es decir, aquellas decisiones sin un impacto directo en los requisitos del sistema y que por tanto pueden ser tomadas en etapas más avanzadas del desarrollo del producto. Todos estos aspectos han surgido durante la implementación de la prueba de concepto de **Circle**.

4.1. Librería común

Una vez comenzada la labor de implementación del primer prototipo de la prueba de concepto, quedó patente la necesidad de compartir código entre distintos componentes del sistema. Este era el caso de algunos objetos de dominio compartidos en varios casos de uso, y especialmente, de las clases que implementan la comunicación entre servicios. Toda esta lógica fue, en las versiones siguientes del prototipo, desplazada a una librería común: **circle-core**. Todos los servicios que conforman el sistema, incluso el API*Gateway*, la incluyen entre sus dependencias. Sus contenidos han sido diseñados en tres categorías fácilmente identificables:

- **Capa de dominio**: objetos relativos al dominio del sistema, compartidos por varios servicios. Por ejemplo, la clase dedicada a encapsular identificadores únicos.

- **Capa de aplicación**: clases que encapsulan la lógica a nivel de servicios de aplicación. Por ejemplo, los objetos que representan las órdenes, consultas o eventos.

- **Capa de infraestructura**: clases que gestionan las comunicaciones entre servicios y encapsulan la implementación de AMQP.

Finalmente, el hecho de manejar una librería común ha sido una muy buena decisión de diseño, ya que ha evitado una gran cantidad de replicación de código.

4.2. GraphQL

Una decisión de implementación que ha traído importantes implicaciones es el uso de GraphQL [10] como protocolo de comunicación entre el cliente y el API *Gateway*. GraphQL es un patrón de diseño de APIs desarrollado por Facebook desde 2012 y que busca sustituir a REST, mejorando algunas de sus limitaciones.

Conceptualmente, GraphQL es un lenguaje de consultas definidas de acuerdo a un modelo de datos expuesto por el servidor. En la práctica, se ejecuta habitualmente sobre HTTP y permite interaccionar con servicios web de una forma mucho más rica que REST. De cara al consumidor de la API, GraphQL expone tres formas distintas de interacción:

- Mediante *queries*, esto es, consultas de datos al servidor. Permite la anidación y concatenación de consultas, además de la posibilidad de seleccionar sólo aquellos campos de las entidades que sean necesarios.

- Mediante *mutations*, es decir operaciones que buscan transformar el estado interno del servidor. Al igual que las consultas, pueden estar parametrizadas y concatenadas, de forma que en un misma petición es posible ejecutar varias de ellas.

- Mediante *subscriptions*. Las suscripciones son un tipo de dato especial de GraphQL, muy similar a la *query*. La principal diferencia pasa porque, con ayuda del cliente GraphQL que las ejecute, pueden actualizar información de forma dinámica. Por ejemplo, es posible realizar una suscripción en un cliente web sobre el precio de un activo que cotiza en bolsa. La suscripción devolverá su valor actual y de forma totalmente autónoma, según el servidor vaya publicando cambios, actualizará la información en el cliente.

Con fines ilustrativos se ha adjuntado en el anexo B un ejemplo de consulta y mutación, junto a sus respuestas.

La decisión de utilizar GraphQL en lugar de REST no viene dada únicamente por su potencia y riqueza semántica. La motivación principal para ello es lo bien alineado que conceptualmente se encuentra con el patrón CQRS. Por un lado, segrega perfectamente entre consultas y órdenes, como exige el patrón. Si bien es cierto que GraphQL fuerza a que las órdenes (mutaciones en este caso) devuelvan también algún valor siempre, esto puede ser útil para retornar

al cliente información básica sobre la aceptación o no de la orden. En el caso de prueba, la ejecución de cualquier *command* retorna siempre un objeto con la interfaz definida en la Figura 5.

```
{
  "data": {
    "signUpUser": {
      "accepted": true,
      "failureReason": null
    }
  }
}
```

Figura 5: Objeto de respuesta de la mutación `signUpUser`.

Sin embargo, donde más brilla la simbiosis entre CQRS y GraphQL es en el uso de suscripciones. Como ya se ha comentado en la sección de arquitectura, una de las principales desventajas del uso de CQRS es lo difícil del manejo de clientes de la API fruto de la ausencia de una respuesta síncrona de las órdenes. Esto, en circunstancias normales, en las que se desee mantener cierta calidad en la experiencia de usuario, lleva habitualmente a mecanismos de *polling*, en los que se ejecuta una encuesta al servicio de forma constante para simular sincronismo. Esta solución, si bien funcional, resulta muy poco elegante y aún menos eficiente.

Para evitar esta aproximación resultan de gran utilidad las suscripciones. Gracias a GraphQL es posible suscribirse a una consulta que devuelva el resultado de un *command*, y, de forma asíncrona, recibir la información resultante en el momento de su completitud, sin romper por ello los principios de CQRS. La forma de lograr este funcionamiento pasa por el uso adicional de eventos de dominio. Una vez completada la ejecución de una orden, esta desencadena la publicación de un evento de dominio notificando este suceso. El API *Gateway*, suscrito a tal evento, resultará notificado cuando esto suceda, y podrá extraer el resultado de la ejecución de la *payload* del evento recibido. Como resultado de la notificación, el API *Gateway* publicará una actualización en la suscripción que el cliente habrá registrado tras enviar el *command*. De esta forma, el cliente podrá recibir el resultado del comando sin necesidad de mecanismos de encuesta o *polling*. El comportamiento descrito puede verse en el diagrama de secuencia de la figura 6.

El esquema de datos de GraphQL para la prueba de concepto se encuentra disponible en el repositorio del código fuente, enlazado en el anexo E.

4.3. Tecnologías

A continuación, se enumeran las tecnologías empleadas en la elaboración de la prueba de concepto, y describimos brevemente aquellos aspectos de las mismas que han impactado en el proyecto. Cabe señalar que la elección de estas

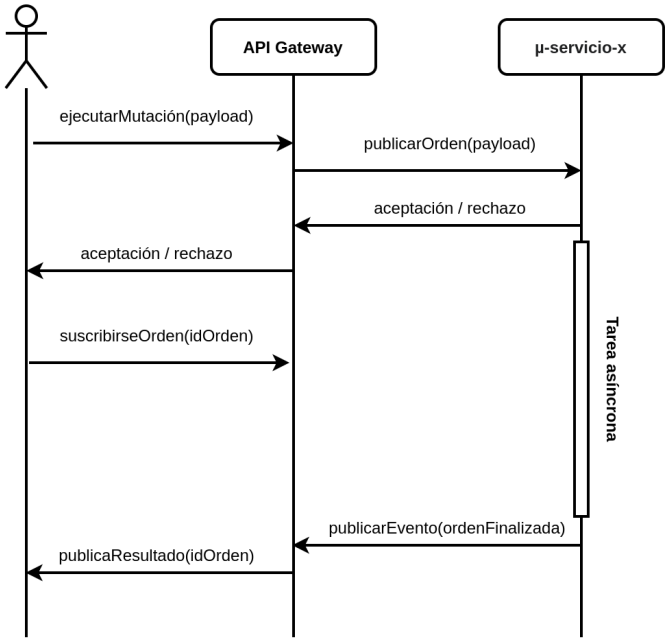

Figura 6: Interacción entre el API *Gateway* y el cliente mediante suscripciones GraphQL

tecnologías, si bien justificadas, distan de ser la única elección posible o válida, ya que en la actualidad, la variedad de herramientas y entornos para el desarrollo de sistemas basados en microservicios es enorme.

4.3.1. NestJS

NestJS [15] es el *framework* de NodeJS escogido para vertebrar los distintos servicios que componen el sistema. Se trata de un *framework* profesional, recomendado para aplicaciones *enterprise* por su gran solidez y respeto por las buenas prácticas de desarrollo de código. Utiliza TypeScript, la versión tipada de JavaScript, el cual a su vez provee de una mayor seguridad y robustez.

Se ha optado por utilizarlo ya que, entre otras ventajas, cuenta con soporte nativo para la inyección de dependencias. Esto resulta especialmente importante en esta prueba de concepto, ya que la comunicación entre capas es muy habitual en el sistema. Mediante la inyección, se consigue mantener un código legible y mantenible y que respeta los principios de diseño de la arquitectura hexagonal [2]. Esto se ha considerado de gran importancia en la prueba de concepto, pues permite centrar todo el esfuerzo del desarrollo en optimizar los mecanismos de comunicación entre servicios sin tener que dedicar apenas esfuerzo a que el código resultante sea de una calidad aceptable.

4.3.2. RabbitMQ

Después de la elección del *framework* de desarrollo, quizá la siguiente decisión tecnológica más importante sea la del *broker* de mensajes. Esta decisión toma aún mayor relevancia en un entorno como el propuesto, donde el paso de mensajes actúa como parte nuclear del sistema.

En el caso de prueba se ha decidido utilizar RabbitMQ [37]. RabbitMQ es un *broker* de mensajes ampliamente utilizado en la industria del desarrollo y proporciona soporte nativo para AMQP [20], el protocolo de paso de mensajes utilizado en el proyecto. La decisión de utilizarlo viene de dada por ser una solución de código abierto, ampliamente utilizada y documentada y con una gran comunidad de desarrolladores detrás. Además, existen multitud de librerías para NodeJS [24] que proporcionan mecanismos de interacción de alto nivel con este *broker*. Entre ellas, *amqp-lib*, la seleccionada para esta tarea en el caso de prueba.

RabbitMQ, además de proporcionar la infraestructura básica de paso de mensajes y gestión de colas da soporte a otras tareas de vital importancia. Por un lado, gestiona de forma transparente la persistencia de los mensajes en las colas. De esta forma, si los nodos de RabbitMQ cayesen, sería posible recuperar el estado de las colas anterior a la caída. De la misma forma, si no hubiese ninguna instancia lista para el consumo de los mensajes en la cola, RabbitMQ se encargaría de almacenarlos hasta que alguna lo hiciese. Por otro lado el *broker* proporciona mecanismos de QoS (*quality of service*) propios, que permiten que el reparto de los mensajes entre los distintos consumidores pueda regirse por unas normas más ricas que un simple mecanismo *round-robin*. En el caso de prueba, estas políticas han sido configuradas mediante una estrategia basada en *ACKs*. Esto es, evitando despachar nuevos mensajes en la cola a los servicios que aún no han confirmado la ejecución de la tarea anterior. Esto evita que por azar varios mensajes de ejecución pesada acaben en una misma instancia, mientras otras permanecen ociosas.

Por último, cabe destacar que RabbitMQ provee de un panel de control gráfico con información sobre carga del sistema, estado de las colas, suscripciones activas, etc. Esta herramienta ha resultado determinante en el proceso de depuración del sistema.

4.3.3. TypeORM

TypeORM [23] ha sido la opción elegida como traductor objeto-relacional (ORM por sus siglas en inglés). Esta pieza de *software*, encapsulada como paquete de NodeJS [24], es la encargada de gestionar la comunicación entre los servicios y sus respectivas bases de datos. Además, facilita la gestión de la persistencia al transformar las entidades almacenadas en la base de datos en objetos de TypeScript [18]. Este proceso es conocido como rehidratación.

La elección de TypeORM viene motivada por ser el ORM más estable, robusto y documentado del ecosistema de TypeScript. Resulta especialmente sencillo de integrar con NestJS, siendo el ORM de referencia del *framework*.

Además, TypeORM implementa el patrón de acceso de persistencia repositorio [11]. Esto implica la provisión de una clase para la interacción de una entidad con la persistencia, en la que se encapsulan todos los métodos necesarios para crear, modificar, obtener y eliminar instancias de la entidad en cuestión. Además, la clase repositorio gestiona de forma totalmente transparente la conversión de las sentencias SQL que ejecutan efectivamente las acciones solicitadas. Teniendo en cuenta que TypeORM crea estas clases de forma totalmente automática, la elección de esta tecnología resulta muy adecuada, al permitir centrar todo el esfuerzo de desarrollo en el ámbito del proyecto.

Resulta interesante recordar llegado este punto el modelo de persistencia propuesto para la prueba de concepto. Cada servicio del sistema maneja un modelo de datos independiente, formado por las entidades que participan en sus casos de uso. Sin embargo, para el conjunto de instancias de un mismo servicio, se comparte una misma base de datos alojada remotamente. Cada instancia mantiene abierta una conexión con dicha base de datos.

Como detalle final de implementación, destacar que el *software* de gestión de base de datos escogido ha sido PostgreSQL [21]. Se trata de una decisión totalmente arbitraria, ya que, para los casos de uso que se plantean, no aporta ninguna diferencia significativa con otros motores.

5. Despliegue y métricas

En esta sección se presentarán los detalles del despliegue de **Circle**, nuestra prueba de concepto. Si bien se describirá el proceso seguido para su despliegue, la esencia de las pautas propuestas servirá como guía para cualquier otra implementación de la "arquitectura socializada".

5.1. Análisis del despliegue

5.1.1. Puesta en marcha de "contenedores"

Por un lado, como ha sido ya anteriormente comentado, para lograr una buena adopción del modelo arquitectural, resulta imprescindible una masa importante de usuarios dispuestos a alojar réplicas del sistema. Si el proceso de instalación resulta complejo, difícilmente se podrá lograr este objetivo. En el despliegue realizado, esto se ha conseguido mediante la "contenerización" de los "servicios socializables". De esta forma, la ejecución de una réplica del sistema es tan sencilla como levantar un contenedor, que en términos prácticos, se traduce en la ejecución de una única orden *shell*. Todas las dependencias, configuraciones y código se encuentran empaquetados dentro sin ninguna necesidad de intervención del usuario, y con garantía de ejecución correcta en cualquier máquina. El único requisito es contar con una instalación de Docker [8], algo sencillo de obtener y bastante habitual hoy en día.

Como sistema de gestión de contenedores ha sido empleado el ya mencionado Docker. Se ha optado por esta solución ya que es la solución de virtualización ligera más popular y por encontrarse su proceso de empaquetado ampliamente documentado.

En términos prácticos, cada servicio ha sido incluido en un contenedor distinto. Para facilitar la distribución de estos, las imágenes de los contenedores han sido alojadas en un repositorio público en Docker Hub[10]. Los tres servicios que componen el sistema pueden ser ejecutados de forma local con las órdenes que aparecen en la Figura 7.

```
docker run -d pitazzo/circle:users-service
docker run -d pitazzo/circle:content-service
docker run -d pitazzo/circle:notifications-service
```

Figura 7: Secuencia de órdenes para levantar los tres servicios del sistema.

Una vez descargadas las imágenes (aproximadamente, de 300MB cada una), comienza de forma automática su ejecución, sincronización con el *broker* de mensajes y enlace con la base de datos, todo ello sin intervención de usuario. Tampoco es necesario, en ningún caso, que el administrador del "sistema socializado" realice ninguna acción de registro. Cabe destacar que es perfectamente válido ejecutar un único servicio, uno de cada o varias instancias del mismo y ninguna de otro.

[10] https://hub.docker.com

Para el proceso de "contenerización" han sido diseñados sendos *Dockerfiles* para los servicios. A modo de ejemplo, se adjunta uno de ellos en el anexo C. El proceso de "contenerización" se ha diseñado buscando minimizar el tamaño y demanda de recursos de los contenedores. Se resume en los siguientes pasos:

1. Instanciación del contenedor de construcción (más pesado al incluir algunas dependencias de compilación).
2. Declaración de variables de construcción.
3. Instalación de dependencias globales de construcción.
4. Copiado del código fuente del servicio y de la librería común.
5. Sustitución de secretos y credenciales.
6. Instalación de dependencias del proyecto.
7. Compilación del sistema.
8. Ofuscación del código ejecutable.
9. Instanciación de la máquina de ejecución (imagen ligera *alpine* con mínimas dependencias).
10. Copiado del código ejecutable ya ofuscado y arranque del mismo.

5.1.2. Ofuscación

Es importante no perder de vista las consideraciones de seguridad que subyacen en la arquitectura propuesta. Por un lado, al alojarse el sistema en una infraestructura totalmente ajena a la organización, se está haciendo entrega directa a los usuarios de todas las reglas de negocio del sistema. Por otro lado, es necesario incluir dentro del ejecutable todos los credenciales necesarios para acceder a la infraestructura crítica del sistema: el *bus* de mensajes y las bases de datos. Ambos aspectos suponen claros riesgos de seguridad.

En lo que se refiere al primer problema de seguridad expuesto, la solución propuesta en el despliegue es la ofuscación [4] del código ejecutable incluido en los contenedores. Esto es, la transformación mediante mecanismos criptográficos del código final, de forma que resulta totalmente incomprensible pero aún así es completamente ejecutable. De esta forma, pasa a ser prácticamente imposible aplicar ingeniería inversa al código y alterar su funcionalidad con éxito. En cuanto al segundo problema, los credenciales, los literales que los almacenan en el código son alterados también mediante operaciones *bitwise* de muy bajo nivel, resultando casi ilocalizables. Como se detalla en el capítulo de seguridad, si bien la ofuscación puede ser una primera aproximación interesante, no resulta suficiente en un sistema "industrial". En dicha sección se proponen algunas alternativas a este proceso.

Para que el proceso de ofuscación ocultase los credenciales de acceso con éxito se ha tenido que abandonar la buena práctica de incluirlos en un fichero

de configuración junto al código fuente. En su lugar, en tiempo de compilación, los literales del código fuente son sustituidos de forma automática, tomados de un fichero externo al código fuente. Esto es así ya que, en caso de haber seguido la aproximación clásica, dicho fichero de configuración hubiera tenido que ser incluido en los contenedores sin ningún tipo de encriptación.

En el anexo D se ha adjuntado un ejemplo de código y literales ofuscados mediante *javascript-obfuscator* [1], la librería utilizada para este proceso durante la "contenerización".

5.2. Alojamiento

En cuanto al alojamiento utilizado para los distintos elementos del sistema, se ha optado por la siguiente infraestructura.

- **Persistencia de datos**. Base de datos PostgreSQL [21] alojada en Heroku [29] en un centro de datos dentro de la Unión Europea.
- ***Broker* de mensajes**. Instancia de RabbitMQ [37] provista por CloudAMQP [20] en un centro de datos dentro de la Unión Europea.
- ***API Gateway***. Entorno de ejecución çontenerizado" provisto por Heroku en EEUU. Provee además de TLS (Transport Layer Security) y nombre de dominio.
- **Servicio de *email***. Servidor de envío de EtheralEmail[11]. Dado que algunas historias de usuario requieren del envío email, se ha utilizado un doble de servidor SMTP[12]. Este doble permite simular el envío de mensajes sin ocasionar grandes volúmenes de mensajes innecesarios.
- **Servicios**. Réplicas distribuidas en cuatro máquinas domésticas con conexiones ADSL[13] y fibra óptica localizadas en España y Francia.

A la hora de realizar pruebas reales sobre el sistema desplegado, debe tenerse en cuenta que las instancias alojadas en Heroku (base de datos y API *Gateway*) se encuentran dentro de un plan gratuito que causa que se suspendan tras 30 minutos sin actividad. Esto implica que tras este periodo, la primera consulte falle, veinte segundos después del fallo, el sistema se comporta con normalidad al haber abandonado el estado de suspensión.

5.3. Métricas analizadas y resultados obtenidos

Para la evaluación del sistema se ha planteado el análisis de tres métricas de rendimiento [14]: tiempo de respuesta, demanda de recursos y latencia. Nuestro interés al analizar estas métricas es únicamente probar la viabilidad del sistema. Aunque ciertamente podrían ofrecer un asesoramiento más completo del sistema

[11] https://ethereal.email/
[12] *Simple Mail Transfer Protocol*
[13] *Asymmetric Digital Subscriber Line*

se ha optado por no profundizar más en ello y no extender el trabajo más allá de sus propósitos. No obstante, en un entorno de desarrollo "industrial" sería necesario un estudio mucho más amplio.

Las condiciones tenidas en cuenta fueron las siguientes:

- Dado el despliegue distribuido planteado, posibilidad de completar con éxito cada una de las historias de usuario.
- Tiempo medio de completitud de cada historia de usuario para 10 peticiones secuenciales.
- Tiempo medio de completitud de cada historia de usuario para 10 peticiones concurrentes.
- Demanda media de memoria y picos de CPU por tipo de servicio.
- Para cada máquina doméstica, sus tiempos de latencia medios hasta la infraestructura común (el *broker* de mensajes).

5.3.1. Tiempos de respuesta

Estos resultados han sido obtenidos mediante: 1) inspección manual para evaluar la completitud, y 2) un servicio de *testing ad hoc* para evaluar los tiempos de respuesta. El mismo se encuentra incluido en el repositorio de código del sistema, documentado en el anexo E.

En cuanto a los resultados de las mediciones, estos han sido recogidos en el cuadro 1. Como puede apreciarse, el tiempo medio de respuesta se encuentra en el entorno de los 600 ms incluso para peticiones concurrentes, un tiempo bastante razonable.

Se ha observado una diferencia de tiempos de respuesta para los casos de prueba concurrentes en función del número de réplicas disponibles, lo cual resulta lógico.

Por otro lado, dada la naturaleza de lenguaje de consulta de GraphQL, los resultados obtenidos varían de forma importante en función de los campos solicitados. En las pruebas realizadas se han solicitado únicamente datos de una misma entidad. Por ejemplo, a la hora de solicitar información de un usuario, no se han solicitado además datos de sus *posts*. Estas consultas anidadas hubiesen implicado la interacción con un segundo servicio, haciendo imposible comparar los resultados de forma homogénea.

Al respecto de las consultas anidadas, se han detectado aumentos de tiempo exponenciales al aplicarlas. Esta situación ha sido especialmente notable en la historia de usuario HU4, en la cual los tiempos ascendían hasta más de 6000 ms. Este comportamiento se relaciona con el hecho de que las consultas anidadas implican la secuencialización del envío de mensajes a los microservicios: siguiendo el ejemplo anterior, hasta no obtenerse la información de los usuarios no se obtienen sus mensajes. Este es el comportamiento estándar de GraphQL y carece de una solución obvia. Por ese motivo, ha sido incluido en la sección de trabajo futuro como una cuestión interesante a plantear.

Código	Éxito	T_{medio} secuencial (ms)	T_{medio} concurrente (ms)
HU1	Sí	412.3	494.6
HU2	Sí	408.1	548.4
HU3	Sí	468.4	600.9
HU4	Sí	458.3	546.2
HU5	Sí	438.6	569.9
HU6	Sí	432.9	547.6
HU7	Sí	407.3	507.9
HU8	Sí	493.4	641.8
HU9	Sí	521.8	633.3
HU10	Sí	416.2	518.1
HU11	Sí	402.0	542.4
HU12*	Sí	-	-
HU13*	Sí	-	-
HU14*	Sí	-	-

Cuadro 1: Tiempos de ejecución secuencial y concurrente de las historias de usuario del sistema

* Estas historias de usuario causan únicamente efectos laterales, por lo que su tiempo de respuesta no puede medirse.

5.3.2. Demanda de recursos

En cuanto a la demanda de recursos de los contenedores en ejecución, los resultados han sido también satisfactorios. El impacto de la ejecución de cada contenedor es mínimo en el sistema, tal como puede verse en el cuadro 2. En cualquier caso, ninguno de los usuarios que participó en la prueba reportó problemas de rendimiento en el equipo aportado.

Servicio	Demanda media RAM (MB)	Pico CPU (%)
users-service	85.0	4.5
content-service	95.0	3.0
notifications-service	110.0	6.0

Cuadro 2: Demanda de recursos de los distintos servicios contenerizados. Muestra obtenida con métricas de Docker.

En el servicio de notificaciones puede observarse una demanda de recursos ligeramente superior. Se deduce que esta circunstancia es así debido a que este servicio, a diferencia de los demás, interacciona con el servicio externo de correo.

5.3.3. Tiempos de latencia

En cuanto a los tiempos de latencia, se ha calculado una tiempo de acceso medio entre las réplicas y el *broker* de mensajes de 35 ms. Dado el alto volumen de mensajes que réplicas y *broker* intercambian resulta especialmente importante que esta cifra sea lo más baja posible, siendo uno de los factores más críticos para el rendimiento del sistema creemos que podemos estar satisfechos con el resultado.

6. Riesgos de seguridad

Tras un primer análisis de la "arquitectura socializada", es probable que los primeros argumentos que se nombren en su contra estén relacionados con riesgos de seguridad. Esto es absolutamente razonable ya que, desde el momento en el que el entorno de ejecución es tan hostil como una máquina ajena y desconocida, los posibles vectores de ataque se multiplican. En esta sección se discutirán algunos de los riesgos derivados de las características del modelo arquitectural.

6.1. Gestión de credenciales

Uno de los riesgos más evidentes de la "arquitectura socializada" es la gestión de credenciales de acceso de los servicios a la infraestructura del sistema. Específicamente, los credenciales de acceso la base de datos y a los *buses* de órdenes, consultas y eventos.

En modelos arquitectónicos tradicionales esto no supone un especial riesgo de seguridad: los entornos de ejecución son privados y es seguro almacenar todo tipo de secretos en ficheros locales de configuración. Mientras no se comprometa la seguridad de la máquina (algo complicado, al tener control sobre ella y poder aplicar cualquier configuración de seguridad deseada), los credenciales permanecerán a salvo.

Esto deja de ser así en el momento en que el entorno de ejecución es una máquina ajena y desconocida. En ningún caso los credenciales de acceso a infraestructura tan crítica deben ser accesibles por el anfitrión, mucho menos por un potencial atacante. Se trata de una problemática similar a la que experimentan las *single page applications* (SPA) al ser ejecutadas en el navegador del cliente: de incluirse algún secreto en su código fuente, este podría ser fácilmente comprometido.

Una primera solución *naive* a este problema podría ser especificar directamente los credenciales de acceso en el código fuente del servicio y posteriormente ofuscar el código de este mediante alguna de las múltiples herramientas disponibles a estos efectos [1]. Si bien esta solución podría ser suficiente para aplicaciones de baja criticidad, y haber sido la aplicada en la prueba de concepto, está lejos de proveer una seguridad sólida. Un análisis forense de la aplicación y su entorno de ejecución, si bien muy complejo, hacen totalmente factible obtener los credenciales de acceso.

La solución que se plantea para esta problemática es similar a la que se propone habitualmente como buena práctica en el desarrollo de SPAs: la creación y despliegue de un servicio externo proveedor de credenciales. Este servicio sería responsable de proveer, en tiempo de ejecución, a cada servicio de unos credenciales generados dinámicamente.

Idealmente, el servicio de credenciales tendría las siguientes características:

- Caducidad de credenciales. Los credenciales de acceso a infraestructura expedidos por el servicio deberán tener una validez breve en el tiempo. Tras su expiración, deberán ser renovados. De esta forma, en caso de comprometerse, la ventana temporal de un posible ataque se minimiza.

- Credenciales ligados a la identidad del usuario que "socializa el servicio". Esto es, expedir credenciales para la base de datos y el *bus* de mensajes de acuerdo a, por ejemplo, una clave que identifica al usuario que va a ejecutar el servicio y formaría parte de la configuración del despliegue. De esta forma, en caso de detectarse un comportamiento malicioso para unos credenciales de infraestructura concretos, bastaría con revocar la clave que identifica al usuario. Así, no podría solicitar nuevos credenciales al servicio proveedor.

- Ligar la expedición de credenciales a que sean solicitados desde determinadas direcciones IP.

- Transmitir los credenciales y sus datos de renovación siempre mediante protocolos seguros que incluyan mecanismos de cifrado.

Cabe destacar que, a pesar de optarse por la solución basada en el servicio proveedor de claves, la ofuscación del código sigue siendo un mecanismo útil para dificultar la obtención de estos.

6.2. Alteración del funcionamiento

Además de la provisión de credenciales existe una segunda problemática de seguridad a tener en cuenta: la posibilidad de la alteración de las reglas de negocio de los "servicios socializados". En principio, no existe ningún impedimento para que un usuario malicioso pueda acceder al código compilado de un servicio y modifique su lógica. Por ejemplo, en el caso de prueba, sería factible modificar el código del servicio de usuarios de forma que al recibir un evento de dominio por una nueva suscripción, en lugar de incrementar en una unidad el número almacenado de suscriptores, lo hiciese en dos unidades.

La primera barrera contra esta problemática es, de nuevo, la ofuscación del código. El código ofuscado es prácticamente imposible de comprender y más aún de modificar con éxito. Además, es posible ejecutar el proceso de ofuscación con opciones adicionales orientas a dificultar aún más la edición. Por ejemplo, es factible ofuscar el código de tal forma que en caso de tabularse para facilitar la edición, este deje de funcionar. A pesar de que estos mecanismos hacen improbable la alteración del funcionamiento de los servicios, no eliminan completamente la problemática: la alteración del código, con recursos suficientes, es posible. Para minimizar en mayor medida este riesgo, existen dos aproximaciones complementarias a la ofuscación.

La primera pasa por asegurar los mecanismos de persistencia. La consecuencia más grave de una alteración de la lógica de negocio sería la persistencia de datos alterados, ya que de lograrse esto, los datos manipulados alcanzarían el flujo normal de ejecución de los demás servicios inalterados. Para evitar esto, existen dos posibilidades:

- Ejecutar comprobaciones de integridad a nivel de base de datos, previas a cualquier persistencia.

- Mapear todos los casos de uso de los servicios en *stored procedures* de la base de datos. Esto es, limitar las posibles interacciones del servicio con la base de datos a una serie de procedimientos almacenados en ella previamente. El acceso a estos procedimientos deberá estar limitado en función de los credenciales expedidos por el servicio de provisión de credenciales. De esta forma, únicamente los servicios debidamente autorizados podrán ejecutar un conjunto reducido de acciones de persistencia.

La segunda aproximación se basa en el uso de funciones *hash* aplicadas sobre el código ejecutable ofuscado. Aplicar una función de *hashing* podría ser un paso previo para la obtención de credenciales. De esta forma, si al obtenerse la firma del código ofuscado se detectase una alteración, se denegaría la concesión de credenciales. Esta operación podría hacerse más segura añadiendo en el código una cadena arbitraria provista previamente por el servicio de credenciales, en un concepto similar al del *challenge* en el apretón de manos SSH. Si bien el uso de una función *hash* dificulta aún más la manipulación del código, esta solución tampoco es perfecta, ya que sería posible manipular el mecanismo de cálculo de la firma del código. A pesar de ello, si este procedimiento también se ejecuta mediante código ofuscado, se aumenta de nuevo la complejidad del ataque.

6.3. Conclusión

De los riesgos de seguridad (y sus soluciones propuestas) se desprende que la "arquitectura socializada" puede ser segura para aplicaciones de baja criticidad. Sin embargo, para aplicaciones propensas a ser atacadas, es necesario un nivel de seguridad adicional que las soluciones propuestas no alcanzan. Por este motivo, será necesario continuar trabajando en el futuro para obtener mecanismos que garanticen un nivel adicional de seguridad.

7. Consideraciones para entornos "industriales"

En este capítulo se analizan los aspectos que han quedado fuera del ámbito de la prueba de concepto, pero que, por su especial relevancia, deberían ser considerados en un entorno "industrial".

7.1. Mecanismos de caché

Una de las primeras optimizaciones que se puede implementar en cualquier sistema informático, y más en arquitecturas de microservicios, es el uso de mecanismos de caché. En el caso de uso propuesto cobran especial relevancia, ya que en un ámbito en el que los tiempos de latencia pueden ser tan variables, una conexión lenta al proveedor de persistencia tendría consecuencias muy importantes en el *throughput* global del sistema. Además, dado el objetivo de la arquitectura de reducir al máximo el impacto de alojar una "réplica socializada", siempre debe perseguirse la reducción del uso del ancho de banda.

Los mecanismos de memoria caché, si bien conceptualmente sencillos (guardar en un soporte de acceso rápido a datos que pueden ser necesarios más adelante y son costosos de obtener) están regidos por unas estrategias que ya resultan complejas en entornos locales. Esta complejidad es mucho mayor cuando el cacheado sucede en entornos distribuidos, como es nuestro caso. La naturaleza distribuida, además de incrementar tiempos de latencia, da lugar a un buen número de casos especiales adicionales. Por ejemplo, cómo gestionar que una réplica modifique un dato que otra había cacheado previamente o cómo invalidar un registro de forma remota. Es precisamente esta complejidad lo que ha motivado que el cacheado distribuido no haya sido considerado en esta prueba de concepto inicial. Sin embargo, se trata de un aspecto absolutamente central en cualquier sistema real que se apoye en la arquitectura socializada.

Para la implementación de este mecanismo existen varias soluciones bien conocidas en la industria, como son Redis [30] y Memcached [3].

7.2. Instalación para usuarios no técnicos

Actualmente, en la prueba de concepto desarrollada, la distribución de las instancias de servicios se realiza mediante imágenes de contenedores Docker, con las particularidades descritas en la sección relativa al despliegue. Esta distribución simplifica en buena medida que usuarios técnicos se presenten a colaborar con una organización ya que para ellos el proceso de instalación consiste únicamente en la ejecución de un comando *shell*.

Sin embargo, esto puede ser demasiado complejo para usuarios no técnicos nada familiarizados con tecnologías de virtualización que, evidentemente, nunca se han visto en la necesidad de desplegar un contenedor. Es por esto que, para poder alcanzar también este segmento de usuarios (numéricamente mayoritario), resultaría imprescindible un proceso de instalación aún más simplificado y al que puedan estar más acostumbrados. Por ejemplo, un instalador auto ejecutable encargado de instalar Docker de forma transparente y ejecutar las imágenes

necesarias. Durante este proceso de instalación se podría solicitar la clave de usuario propuesta en el capítulo de seguridad. El ejecutable debería ser provisto por la organización, por ejemplo, mediante una descarga desde su página web.

7.3. Instrumentación y módulo de control

Con el modelo de distribución actual, las instancias de los servicios, si bien dentro de un contenedor, son ejecutadas directamente en la máquina anfitriona. Aunque esto es óptimo a nivel de prestaciones, tiene importantes contrapartidas. Es imposible ejercer ningún tipo de control sobre las instancias ni obtener información sobre ellas. Tampoco es posible acceder a los registros de ejecución una vez desplegadas. En un entorno real resultaría extremadamente interesante poder realizar acciones como:

- Detener y reiniciar instancias en caso de mal funcionamiento.
- Actualizar imágenes de instancias de forma remota.
- Ejecutar determinadas órdenes de forma remota para realizar comprobaciones puntuales.
- Obtener métricas en tiempo real sobre el rendimiento de la instancia.
- Obtener acceso a los *logs* de la instancia para poder detectar errores y depurarlos.

Para dar soporte a estas funcionalidades resulta imprescindible añadir un nivel de indirección adicional. Resultaría necesario que la imagen de contenedor distribuida fuese, en lugar de correspondiente a un servicio concreto, un módulo de control con posibilidad de levantar contenedores de forma autónoma. Este módulo de control genérico sería responsable de dar soporte a los casos de uso anteriores e incluiría las librerías de instrumentación necesarias para llevarlos a cabo.

En cuanto a que un contenedor Docker pudiese levantar otros contenedores, si bien puede sonar extraño, es una práctica muy habitual. Es conocida *docker-in-docker* (Docker dentro de Docker) y está ampliamente discutida y analizada [25]. El proceso, a grandes rasgos, consiste en parametrizar el primer contenedor respecto al *socket* que controla Docker en la máquina anfitriona. De esta forma, el primer contenedor puede interaccionar con Docker como si de la máquina anfitriona se tratase, siendo capaz de levantar otros contenedores a su mismo nivel.

7.4. Equilibrado de instancias

Una vez logrado el desarrollo del módulo de control explicado en el punto anterior, resultaría extremadamente útil para la organización responsable del sistema la posibilidad de equilibrar el número de instancias de cada servicio de forma remota y automática. Por ejemplo, si se detectase un bajo número de

instancias de un tipo de servicio y alto de otro, el sistema debería poder ser capaz de, autónomamente, ordenar la destrucción del exceso de contenedores y el despliegue de nuevos. Esto, cabe destacar, solo será posible mediante la existencia de un módulo de control.

La posibilidad de equilibrar la cantidad y tipo de instancias unida a la instrumentación de estas da pie a la configuración de políticas muy ricas. Por ejemplo, sería posible detectar posibles picos de demanda en ciertos servicios, e incluso prevenirlos con antelación. De la misma forma, podrían detectarse valles en la demanda que hiciesen algunas instancias innecesarias, pudiendo estas destruirse de forma remota.

Por último, y en un plano más teórico, sería posible sincronizar el despliegue y destrucción de instancias con la ubicación de los usuarios en un determinado momento. De esta forma, cuando se detectase un pico de usuarios en una determinada región (por ejemplo, por ser ahí de día), podría ordenarse levantar réplicas geográficamente cerca para intentar mitigar las latencias.

7.5. Respaldo en *cloud*

Si bien el uso del *cloud* choca con la justificación de la "arquitectura socializada", la nube podría ser un aliado interesante para garantizar la estabilidad del sistema. Si de acuerdo a los puntos anteriores se instrumentan las instancias y se es capaz de detectar picos de demanda, en supuestos en los que no haya usuarios disponibles suficientes, la nube puede ser la solución.

Dada su naturaleza elástica resultaría totalmente factible que el servicio de equilibrado de carga, una vez detectase la imposibilidad de satisfacer la demanda de recursos, provisionase varias máquinas virtuales *cloud*. Estas acciones podrían ser realizadas de forma programática interactuando directamente con las APIs de los proveedores *cloud*[14] idealmente mediante abstracciones de más alto nivel, como por ejemplo, Terraform[15], un gestor de infraestructura como servicio (IaaS[16]).

[14] https://cloud.google.com/apis/docs/overview
[15] https://www.terraform.io/
[16] *Infraestructure as a service*

8. Cuestiones de viabilidad

En esta sección se comentarán brevemente las principales cuestiones no técnicas que condicionarán la viabilidad de la "arquitectura socializada".

8.1. Número de usuarios

Dado que la esencia de la "arquitectura socializada" es la distribución de la carga de trabajo de un sistema entre los usuarios del mismo, el número de usuarios dispuestos a participar del proceso de socialización resulta uno de los elementos más importantes, si no el más, para la viabilidad de la arquitectura.

Si bien conceptualmente la idea de la "arquitectura socializada" es sencilla, para personas ajenas al mundo técnico puede ser difícil de comprender. Esta incomprensión puede terminar por ser uno de los factores que más limiten la voluntad de los usuarios por compartir parte de sus recursos computacionales. Por este motivo, las organizaciones interesadas en trabajar con la "arquitectura socializada"deberán hacer una importante labor divulgativa entre sus usuarios explicando cómo y para qué participar en la socialización.

Resultará probable que los usuarios se encuentren más dispuestos a colaborar con proyectos *open source* o que persigan algún tipo de bien común que con aquellos con ánimo de lucro, ya que alojar un servicio podría entenderse entonces como "ayudar a cambio de nada" a una empresa. Para que en estos casos la arquitectura socializada también tenga cabida, podría optarse por un modelo de socialización en el cual el número de horas que se aloja un servicio brinde algún tipo de beneficio. Por ejemplo, para modelos de negocio basados en la suscripción podría ofrecerse una rebaja económica en la cuota en función del número de horas de cómputo aportadas. En modelos de negocios basados en publicidad podría optarse por ocultar los anuncios.

En cualquier caso, vincular la socialización a recompensas dentro del dominio exigiría considerar una serie de servicios adicionales que relacionasen la gestión de estas recompensas y la asociación del cómputo con usuarios concretos del sistema. Esto, en principio, queda fuera del alcance de la "arquitectura socializada".

8.2. Adopción de la arquitectura

Otro punto importante a considerar en cuanto a la viabilidad de la arquitectura es su propia adopción. Para que realmente pueda constituir una alternativa seria al *cloud* se tiene que buscar una masa crítica de organizaciones dispuesta a adoptarla. De otra forma, difícilmente podrá subsistir como tecnología nicho. Esto es así por dos motivos:

- Implementar nuevos sistemas que la apliquen generará nuevo conocimiento y soluciones que posibilitarán el desarrollo de futuros sistemas mejorados.
- Resultará mucho más sencillo que un usuario esté dispuesto a alojar un servicio si alojar servicios es algo relativamente habitual en su entorno.

De otra forma, podría ser visto como una extravagancia o incluso algo malicioso.

8.3. Patrocinio institucional

Por último, otra cuestión importante de viabilidad es la disposición de las instituciones a alojar servicios socializados. No hay que olvidar que aunque a priori el alojamiento más habitual para servicios socializados serán máquinas domésticas, existen otras opciones institucionales. Por ejemplo, la cesión de alojamiento en centros de datos de universidades o empresas a determinadas causas o si se cumplen una serie de requisitos (empresas de nueva creación, organizaciones con impacto social positivo, etc).

Otra alternativa de apoyo institucional podría ser instalar servicios socializados en las estaciones de trabajo de los empleados o de los laboratorios docentes. De esta forma, con ningún coste más allá del energético, instituciones públicas y privadas de todo tipo podrían causar un impacto positivo y mejorar su imagen corporativa.

9. Trabajo futuro

En este capitulo se detallan algunas de las cuestiones que bien por su complejidad, bien por el ámbito limitado del proyecto no se han abordado. Estas cuestiones podrían ser objeto de investigación posterior como continuación de este proyecto.

9.1. Autenticación de usuarios

Una cuestión que ha quedado excluida de la prueba de concepto es la de la autenticación de usuarios en el sistema. En el caso de prueba, cualquier usuario puede realizar cualquier acción. En los casos en los que se necesita identificar al usuario (por ejemplo, para publicar un *post*) su identificación es un parámetro más de la operación. Esto no debería ser así. Lo ideal hubiese sido plantear un servicio de autenticación.

Sin embargo, el funcionamiento de este servicio no es una cuestión tan evidente como podría parecer y arroja varias preguntas. ¿La autenticación debería suceder en el API *Gateway* o ejecutarse en cada servicio? ¿Debería agruparse la autenticación en un servicio aparte o formar parte del servicio de usuarios? ¿Debería poder socializarse el servicio de autenticación? ¿Qué implicaciones de seguridad tendría esto?

Como puede desprenderse de estas cuestiones, se trata de una cuestión compleja que bien merece ser analizada mediante una segunda prueba de concepto.

9.2. Optimización de consultas

Como se comenta en la sección correspondiente a resultados de las pruebas, se ha identificado un importante cuello de botella en la resolución de consultas GraphQL anidadas. El problema sucede al generarse dependencias entre los datos de salida de la consulta. Sin embargo, existen supuestos en los que esa dependencia podría evitarse si la consulta anidada recibiese inmediatamente los parámetros de la consulta padre.

Resultaría interesante abordar este problema en el futuro, ya que paralelizar ambas consultas supondría una importante optimización. Además, podrían buscarse pautas de diseño de consultas para minimizar los casos en los que esta paralelización no es posible. Cabe destacar que esta problemática no es exclusiva de la "arquitectura socializada", sino de todas las arquitecturas de microservicios que se exponen mediante una API GraphQL.

9.3. Uso de suscripciones GraphQL

Tal como se enuncia en el capítulo de implementación, una de las grandes ventajas del uso de GraphQL es la posibilidad de usar suscripciones para recibir resultados de *commands* de forma asíncrona en el cliente. Esto permitiría prescindir de los mecanismos típicos de *polling* en clientes que consumen servicios CQRS.

Dado que la prueba de concepto no incluía un cliente, estos mecanismos se han omitido, pero podría resultar de gran interés implementarlos. Esta implementación traería un buen número de cuestiones de diseño a analizar: ¿Debería ofrecerse una única *subscription* para consultar el resultado de cualquier orden dado un identificador del *command*? ¿O en su lugar una por cada comando? ¿Cómo se gestionarían los comandos fallidos?

9.4. Validación de órdenes y consultas en el *API Gateway*

En la implementación actual, la validación de los parámetros de las órdenes recibidas por el sistema se realiza en los propios servicios, como paso previo a su aceptación, emisión del mensaje `CommandResult` de respuesta y ejecución. Se ha seguido este planteamiento ya que esta es la forma de validar los *commands* que dicta CQRS: deben ser los propios `CommandHanlders` los que validen los objetos-valor de los parámetros de la orden como paso previo a la invocación del caso de uso ejecutor.

Sin embargo, dada la naturaleza altamente distribuida del sistema, podría resultar interesante romper esta norma, haciendo que el API *Gateway* fuese el responsable de validación. De optarse por esta aproximación la arquitectura contaría con dos ventajas adicionales. Por un lado, se acortarían los tiempos de respuesta de las órdenes, ya que una vez validadas en el API *Gateway* podría enviarse al cliente el `CommandResult` inmediatamente sin esperar a los tiempos de latencia de la comunicación entre servicios. Por otro, se podría sacar más partido al *broker* de mensajes ya que, en el momento de recibirse el *command* dejaría de ser necesaria la disponibilidad de servicios capaces de procesarlo. De no haber disponibles servicios capaces de ejecutar la orden, el *broker* se limitaría a persistir el mensaje hasta que, eventualmente, un servicio capaz de procesarlo se incorporase a la red. Con este planteamiento, el sistema podría aceptar peticiones de clientes fuese cual fuese la disponibilidad de réplicas, algo muy interesante en el contexto de la "arquitectura socializada".

9.5. Persistencia distribuida

Como última propuesta de investigación futura se encuentra la cuestión de la persistencia distribuida. En la justificación del proyecto se nombraba como motivación importante la recuperación de la soberanía de los datos. Sin embargo, con el modelo actual de "arquitectura socializada" se sigue dependiendo de soluciones *cloud* para almacenar de forma fiable el estado del sistema.

La alternativa es complicada, ya que distribuir la persistencia trae consigo un tremendo número de cuestiones a considerar. A pesar de ello, es una cuestión primordial para lograr los objetivos últimos de la arquitectura y por tanto debería ser abordada. Algunas de las opciones que se proponen para resolver la problemática son las siguientes:

- Socializar la persistencia junto a las réplicas en máquinas ajenas, gestionando de forma clásica las transacciones distribuidas y cuestiones como el consenso con algoritmos bien estudiados, por ejemplo, Raft [22].

- Optar por opciones de alojamiento descentralizadas, como por ejemplo, el Interplanetary File System (IPFS) [5], que provee mecanismos de persistencia totalmente descentralizados.
- Considerar opciones menos habituales como el uso de *blockchain* a modo de *distributed ledger*, con todas sus consecuencias asociadas.

En cualquier caso, deberán ser consideras las implicaciones de seguridad, rendimiento y fiabilidad de la solución planteada, así como su viabilidad técnica.

10. Conclusiones finales

El proyecto puede considerarse exitoso en el momento que ha cumplido su objetivo de enunciar y analizar en profundidad la "arquitectura socializada". Se ha justificado su necesidad en base a la situación de oligopolio del mercado *cloud* y se ha enunciado una propuesta concreta como alternativa. Para ponerla a prueba, se ha diseñado un caso de prueba bien definido y se ha implementado utilizando tecnología moderna. Este caso de prueba ha sido sometido a *testing* de una forma sistemática en un entorno real y se ha analizado su rendimiento. Por último, del proceso de implementación y prueba se han extraído conclusiones relativas a seguridad, implicaciones en entornos reales y viabilidad. Finalmente, se ha propuesto trabajo a realizar en el futuro como continuación de esta investigación.

En cuanto al resultado de la prueba de concepto, puede considerarse satisfactorio. Se considera exitoso al concurrir las siguientes circunstancias:

- Una vez conocida la tecnología, es aceptablemente complejo desarrollar aplicaciones respetando los requisitos de la arquitectura.
- El impacto de alojar "réplicas socializadas" es mínimo en las máquinas anfitrionas: apenas 90 MB por instancia y picos de CPU del 4 %.
- Los tiempos de respuesta en entorno real, si bien altos, entran dentro de lo aceptable con una media de 500 ms.

A pesar de ello, existe multitud de trabajo pendiente a realizarse antes de poder plantear el uso de la arquitectura en entornos reales. En especial, es muy necesaria la investigación adicional en materia de seguridad y tolerancia a fallos.

Este Trabajo Fin de Grado persigue un objetivo realmente ambicioso: intentar reducir el poder del oligopolio de las empresas dedicadas al alojamiento en la nube. Busca hacerlo no desde la regulación ni la confrontación sino intentando ofrecer el germen de una alternativa rentable. Evidentemente, el modelo de "arquitectura socializada" presentado en esta prueba de concepto no puede en ningún caso competir con el ecosistema *cloud*, ni en prestaciones, ni en tiempo de desarrollo, ni en fiabilidad. A pesar de ello ojalá sirva, al menos como inspiración, en el camino de garantizar la libertad en Internet.

Referencias

[1] Tiago Serafim and Timofey Kachalov. JavaScript Obfuscator Tool. `https://obfuscator.io`, 2020.

[2] Alistair Cockburn. The Pattern: Ports and Adapters ("Object Structural"). `https://alistair.cockburn.us/hexagonal-architecture/`, 2020.

[3] Brad Fitzpatrick. Memcached – A distributed memory object caching system. `https://memcached.org`, 2020.

[4] Finn Brunton and Helen Nissenbaum. *Obfuscation: A User's Guide for Privacy and Protest*. The MIT Press, 2015.

[5] Y. Chen, H. Li, K. Li, and J. Zhang. An improved p2p file system scheme based on ipfs and blockchain. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2652–2657, 2017.

[6] Louis Columbus. 83 % of enterprise workloads will be in the cloud by 2020. `https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/`, Jan 2018.

[7] Datometry Inc. Cloud data warehousing survey: Cio insights. `https://datometry.com/resources/surveys/cloud-data-warehousing-survey/`, Jul 2020.

[8] Docker Inc. Docker – Empowering App Development for Developers. `https://www.docker.com`, 2020.

[9] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[10] Facebook. GraphQL - A query language for your API. `https://graphql.org`, 2012–2020.

[11] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[12] Fundación Vodafone. DreamLab. `https://www.vodafone.com/dreamlab/spain`.

[13] Y. S. Hong, J. H. No, and S. Y. Kim. Dns-based load balancing in distributed web-server systems. In *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, pages 4 pp.–, 2006.

[14] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[15] Kamil Mysliwiec. NetsJS - A progressive Node.js framework for building efficient, reliable and scalable server-side applications. `https://nestjs.com`, 2017–2020.

[16] A. Kumar. *Cqrs (Command Query Responsibility Segregation).* Independently Published, 2019.

[17] Martin Fowler. CQRS. `https://martinfowler.com/bliki/CQRS.html`, July 2011.

[18] Microsoft. TypeScript – Typed JavaScript at Any Scale. `https://www.typescriptlang.org`, 2012–2020.

[19] Sam Newman. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, 1st edition, February 2015.

[20] OASIS. AMQP – Advanced Message Queuing Protocol. `https://www.amqp.org`, 2020.

[21] Regina O. Obe and Leo S. Hsu. *PostgreSQL: Up and Running A Practical Introduction to the Advanced Open Source Database.* O'Reilly Media, Inc., 2nd edition, 2014.

[22] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *USENIX*, pages 305–320, 01 2014.

[23] Open source – supported by sponsors. TypeORM – Object-Relational Mapping. `https://typeorm.io/#/`, 2020.

[24] OpenJS Foundation. NodeJS. `https://nodejs.org/es/`, 2020.

[25] J Petazzo. `https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/`, 2020.

[26] Robert Richards. *Representational State Transfer (REST) in Pro PHP XML and Web Services*, pages 633–672. Apress, Berkeley, CA, 2006.

[27] Rob van der Meulen. Understanding cloud adoption in government. `https://www.gartner.com/smarterwithgartner/understanding-cloud-adoption-in-government/`.

[28] Peter Rodgers. Service-oriented development on netkernel-patterns, processes & products to reduce system complexity. *CloudComputingExpo. SYS-CON Media*, 2005.

[29] Salesforce.com. Heroku – Cloud Application Platform. `https://www.heroku.com`, 2020.

[30] Salvatore Sanfilippo. Redis – In-memory data. https://redis.io.

[31] A.S. Tanenbaum and M. Van Steen. *Sistemas distribuidos: Principios y paradigmas*. Pearson Educación, 2008.

[32] Hugh Taylor, Angela Yochem, Les Phillips, and Frank Martinez. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Addison-Wesley Professional, 1st edition, 2009.

[33] ThoughWorks Inc. ThoughWorks: A Global Software Consultancy. https://www.thoughtworks.com, 2020.

[34] Twitter. Twitter – Red social. https://twitter.com/?lang=es, 2020.

[35] University of California at Berkeley. SETI@home. https://setiathome.berkeley.edu, 2020.

[36] Cristina Vargas. Cloud market share report: Aws vs azure vs google cloud 2019: Mcafee. https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/, Jan 2020.

[37] VMware Inc. RabbitMQ – Messaging that just works. https://www.rabbitmq.com, 2007-2020.

Índice de figuras

A. Documentación de puertos

Toda la interacción entre servicios del sistema se realiza mediante el paso de mensajes haciendo uso del protocolo AMQP. Siendo este el principal elemento de especificación de los puertos, resulta también importante la forma en la que estos mensajes se estructuran.

Cada mensaje enviado en el sistema se caracteriza por dos factores. Por un lado, su clave de enrutado, que posibilita que otros servicios se suscriban a su publicación. Por otro, su carga o *payload*. Esta carga consiste en un objeto serializado en función del tipo de mensaje (orden, consulta o evento), en cuyo interior residen los datos necesarios para su procesamiento. Los tres tipos de mensaje derivan de uno común, la *Operation*. A modo ilustrativo, se incluye en la figura 8 la implementación concreta para los mensajes de tipo Orden.

```
export class Command extends Operation {
  constructor(
    id: string,
    attributes: Record<string, any>,
    extraMetadata: Record<string, any>
  ) {
    super(id, "command", attributes, extraMetadata);
  }
}
```

Figura 8: Clase `Command`

Como puede apreciarse, cada comando se caracteriza por un identificador (que coincide con la calve de enrutado), una serie de atributos propios (por ejemplo, los datos del usuario) y un campo opcional para añadir metadatos.

Un ejemplo concreto de un objeto `Command` serializado está adjunto en la figura 9

```
{
  "id": "circle.gateway.1.command.user.signedup",
  "type": "command",
  "ocurredOn": "2020-09-22T16:42:39.469Z",
  "attributes": {
    "username": "pedro",
    "email": "hola@pedromalo.dev"
  },
  "meta": {}
}
```

Figura 9: Objeto de tipo `Command` serializado

B. Ejemplo consulta y mutación GraphQL

Con fines ilustrativos, se adjunta un ejemplo real de consulta (figura 10) y mutación (figura 11) GraphQL junto a sus respectivas respuestas.

```
query user($username: String!){
  user(username: $username) {
    id
    username
    email
    enrollmentDate
    posts {
      id
      title
      body
    }
  }
}
```

```
{
  "data": {
    "user": {
      "id": "5beb69ab-b87f-4b24-aa45-1d6d8c8216d5",
      "username": "pitazzo",
      "email": "me@pitazzo.dev",
      "enrollmentDate": "2020-09-22T16:46:49.969Z",
      "posts": [
        {
          "id": "90750509-bd19-47dc-89ba-e1f83e10b547",
          "title": "Hola mundo",
          "body": "Lorem ipsum dolor..."
        }
      ]
    }
  }
}
```

Figura 10: Consulta GraphQL `user` junto a su respuesta

```
mutation EditUser($editUserInput: EditUserInput!) {
  editUser(editUserInput: $editUserInput) {
    accepted
    failureReason
  }
}

{
  "data": {
    "editUser": {
      "accepted": true,
      "failureReason": null
    }
  }
}
```

Figura 11: Mutación GraphQL `editUser` junto a su respuesta

C. Ejemplo de *Dockerfile*

En la figura 12 se ha adjuntado el *Dockerfile* que construye el servicio de usuarios. Se ha omitido algunos comandos encadenados de sustitución de secretos por claridad.

```
FROM node:14 AS builder

ARG RABBITMQ_URL
ARG DB_HOST
ARG DB_PORT
ARG DB_USER
ARG DB_PASSWORD
ARG DB_DATABASE

WORKDIR /app/circle-core
RUN npm install -g rexreplace typescript javascript-obfuscator
COPY ./circle-core .
RUN rr RABBITMQ_URL $RABBITMQ_URL
    ./src/infraestructure/amqp-service/amqp-service.ts
RUN npm install && tsc
RUN javascript-obfuscator dist --output obfuscated
    && rm -rf dist && mv obfuscated/dist .

WORKDIR /app/users-service
COPY ./users-service .
RUN rr DB_HOST $DB_HOST ./src/app.module.ts && ...
RUN npm install
RUN npm run build
RUN javascript-obfuscator dist --output obfuscated
    && rm -rf dist && mv obfuscated/dist .

FROM node:14-alpine
WORKDIR /app
COPY --from=builder /app/users-service/dist ./users-service/dist
COPY --from=builder /app/users-service/package.json ./users-service/package.json
COPY --from=builder /app/users-service/node_modules ./users-service/node_modules
COPY --from=builder /app/circle-core/node_modules ./circle-core/node_modules
COPY --from=builder /app/circle-core/package.json ./circle-core/package.json
COPY --from=builder /app/circle-core/dist ./circle-core/dist
COPY --from=builder /app/users-service/.env ./users-service/.env
WORKDIR /app/users-service
CMD ["npm", "run", "start:prod"]
```

Figura 12: *Dockerfile* del servicio de usuarios

D. Ejemplo código ofuscado

Con fines ilustrativos, se adjunta en la figura 13 una pequeña porción de código. La versión ofuscada de este código haciendo uso de Javascript Obfuscator [1] se encuentra accesible en `https://pastebin.com/dEKcsEAx`.

```
function hi() {
  console.log("Hello World!");
}
hi();
```

Figura 13: Pequeña pieza de código original

El ejemplo resultante ha sido omitido por cuestiones de espacio. Puede comprobarse de forma manual que el código resultante continúa siendo totalmente funcional y ejecutable.

E. Entrega del producto

El resultado funcional de la prueba de concepto puede ser obtenido de tres formas distintas, en función de los objetivos que se persigan.

- Para revisión del código fuente y compilación manual de sistema, todos los elementos y servicios que los conforman están accesibles en un repositorio público en Github. Este repositorio está disponible en `https://github.com/pitazzo/circle` e incluye una guía mínima sobre su puesta en marcha. Téngase en cuenta que una construcción manual necesitará provisionar una base de datos PostgreSQL y un clúster RabbitMQ.

- Para la ejecución de local de los contenedores, se recomienda hacer uso de las imágenes alojadas en DockerHub en `https://hub.docker.com/repository/docker/pitazzo/circle`. Estas imágenes contienen los servicios que conforman el sistema y están configuradas para interactuar directamente con el sistema real, por lo que al ejecutarlas se participa del proceso de socialización.

  Para la puesta en marcha de los contenedores, una vez se cuenta con una instalación de Docker, basta con ejecutar los comandos de la figura 14.

  ```
  docker run -d pitazzo/circle:users-service
  docker run -d pitazzo/circle:content-service
  docker run -d pitazzo/circle:notifications-service
  ```

  Figura 14: Órdenes necesarias para arranque de los contenedores.

- Para usar el sistema desde la perspectiva del cliente se recomienda interaccionar directamente con la réplica del API *Gateway* alojada en Heroku. Es accesible en el *endpoint* y para la interacción se recomienda el uso de un cliente GraphQL.

  Nótese que será necesario levantar al menos una réplica de cada contenedor de forma local para que el sistema funcione. Así mismo, téngase en cuenta que el alojamiento en Heroku es gratuito, lo que causa la suspensión del API *Gateway* cada 30 minutos. Esto lleva a que en caso de interacción con él, la primera petición resulte errónea hasta que la máquina abandone la suspensión.

F. Glosario de acrónimos

A continuación aparecen los acrónimos utilizados en esta Memoria. La mayoría de ellos ya han sido definidos en el texto y se incluyen aquí a modo de resumen, con la intención de facilitar al lector su consulta.

ADSL	Asymmetric Digital Subscriber Line
AMQP	Advanced Message Queuing Protocol
API	Application Program Interface
CQRS	Command Query Responsibility Segregation
CRUD	Create, Read, Upadate, Delete
DDD	Domain-Driven Design
DNS	Domain Name System
EDA	Event Driven Architectures
FIFO	First In First Out
IaaS	Infraestructure as a service
ISP	Internet Service Provider
MSA	Micro Service Architecture
NAT	Network Address Translation
QoS	Quality of Service
REST	REpresentational State Transfer
SLA	Service Level Agreement
SMTP	Simple Mail Transfer Protocol
SPA	Single Page Applications
SOA	Service Oriented Architecture
TLS	Transport Layer Security